MASTER EN SCIENCES INFORMATIQUES

Conception et développement d'une architecture multi-tiers synchrone / asynchrone

Coyette, Laurent

Award date:
2002

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique.
Année académique 2001 - 2002

Conception et développement d'une architecture multi-tiers synchrone / asynchrone.

Laurent Coyette

Mémoire présenté en vue de l'obtention du grade de Licencié en Informatique.

LBS 9138004

Résumé

Ce travail consiste en la conception et le développement d'une architecture middle-tier dans un environnement Delphi / TCP/IP.
Le nom donné à cette architecture est: R.S.O. (Remote Server Objects).

Les caractéristiques principales de cette architecture sont les suivantes :

- Accès synchrones et asynchrones des objets distants.
- Système performant.
- Support d'un grand nombre de connexions simultanées.
- Concept de ResultSet (emprunté aux bases de données) complètement intégré.
- Accès aux méthodes des objets distants sans définition préalable d'une interface.

Un état de l'art dans ce domaine est d'abord présenté ainsi qu'une description plus précise de produits concurrents tel que CORBA, RMI et EJB.
Viennent ensuite, la spécification, la conception, l'implémentation et les tests de cette architecture.
L'implémentation n'est pas complète. Les accès asynchrones et la gestion de la sécurité n'ont pas été développés.
En ce qui concerne les tests, une comparaison des performances obtenues entre RSO, CORBA et RMI est réalisée.

Mots clés: Delphi, TCP/IP, Middle-Tier, MiddleWare, RSO, CORBA, RMI, EJB, Architecture, Synchrone, Asynchrone, Client-Serveur, 3-Tiers

Abstract

This work consist in design and development of a middle-tier architecture in a Delphi / TCP/IP environment.
The name give to this architecture is R.S.O. (Remote Server Objects).

The main features of this architecture is:

- Synchronous and asynchronous access to remote objects.
- Outstanding system
- High number of simultaneous connection support.
- Result Set concept (borrowed from databases) completely integrated.
- Access to remote objects method without interface definition.

A state of the art in this domain is presented and also a much more precise description of concurrent products as CORBA, RMI and EJB.
Next, come the specification, design, implementation and tests of this architecture.
The implementation is not complete at all. Asynchronous access and security management has not been developed.
Concerning tests, a performance comparison between RSO, CORBA and RMI has been done.

Keywords*:* Delphi, TCP/IP, Middle-Tier, MiddleWare, RSO, CORBA, RMI, EJB, Architecture, Synchronous, Asynchronous, Client-Server, 3-Tiers

Avant-propos

Je remercie tout le corps professoral de l'institut d'informatique et en particulier mon promoteur, Monsieur V. Englebert qui me fut de très bon conseil.

Je tiens aussi à remercier les membres de ma famille et amis pour leur soutien et leur aide.

Enfin, je remercie Monsieur P. Arrotin pour ses idées et conseils.

Table des matières

I. Problématique

A. Contexte

Ce travail intervient dans un contexte assez particulier qu'est celui de la Police Judiciaire. L'informatique de la PJ a été initialement mise en place il y a de cela une dizaine d'années par quelques péjistes éclairés. Cela était tout à fait suffisant pour l'époque. Depuis, l'informatique a bien évolué et est devenue incontournable. Surtout, dans un cadre policier où, l'information est la matière première de l'enquêteur.

Malheureusement, durant ces 10 ans, le service informatique n'a pas su évoluer comme il aurait du. Les raisons sont multiples. La direction n'a peut-être pas attaché assez d'importance à celui-ci, le service lui-même n'a peut-être pas su se vendre.

Toujours est-il que, à l'heure actuelle, le service ne compte qu'une dizaine de personnes, y compris les secrétaires. Ces personnes s'occupent, de la réalisation de cahier des charges pour l'achat de logiciels et de matériels, du développement de nouvelles applications, de la maintenance des applications existantes.
Le service est néanmoins épaulé par des coordinateurs informatiques présents dans chaque brigade. Ces personnes jouent le rôle de « Helpdesk » de première ligne.

Du point de vue du matériel, le tableau est moins catastrophique. En général, il est souvent plus simple, à l'Etat, d'obtenir des dépenses uniques plutôt qu'un budget pour engager du personnel (Même si ces dépenses dépassent largement le budget en question)

Or, les demandes de nos enquêteurs se font de plus en plus nombreuses. Effectivement, le crime ne cesse d'évoluer, des organisations impressionnantes se mettent en place. Si nous voulons pouvoir faire face à ce crime organisé, nous devons disposer d'un outil informatique performant.

La police judiciaire en chiffres :

La police judiciaire compte quelques 2300 personnes réparties, pour deux tiers, en personnel de terrain (les judiciaires) et pour le reste, en personnel administratif.
Plus ou moins 1000 PC sont répartis sur l'ensemble du territoire au sein des diverses brigades de Police Judiciaire.

Ces données risquent, à terme d'être inexactes de par la mise en place de la Police Fédérale.

B. Rappels technologiques

Pour bien comprendre les notions évoquées dans ce document, quelques rappels s'imposent…

Le modèle de programmation le plus couramment utilisé depuis des années, dès lors qu'un programme doit accéder à des données pour les présenter à un utilisateur se nomme **Client/Serveur**.

Ce modèle se compose d'un programme client, d'un programme serveur et d'un moyen de communication entre ces deux programmes.
Le client joue le rôle de l'interface avec l'utilisateur finale. Lorsque le client a besoin d'un service particulier (récupération de données en provenance d'une base de données), il contacte le serveur.

Ce modèle a évolué au fil des années. De nouvelles déclinaisons de celui-ci sont apparues.

1. Les architectures Client/Serveur One-Tier (monolithic)

Ce modèle est apparu à l'époque des mainframes. Un seul programme fonctionnant sur le host constitue le client et le serveur. L'utilisateur utilise alors un terminal connecté au mainframe.

2. Les architectures Client/Serveur Two-Tier

Le client communique alors directement avec un serveur de base de données. La logique applicative se trouve sur le client et sur le serveur (trigger, stored procedure, contraintes d'intégrités, …).

Ce modèle est apparu fin des années quatre-vingts, et est toujours largement utilisé de nos jours.
L'architecture two-tier peut se décliner sous deux formes.

- *Fat clients. (Clients lourds)*

  Dans ce modèle, la logique applicative est implémentée, presque exclusivement, sur le client. Cette solution permet un développement relativement aisé. Le client effectue des requêtes SQL, reçoit les réponses, les traite et enfin, les présente à l'utilisateur.

  L'inconvénient majeur de cette technique est que plus les traitements sont complexes plus les machines clients doivent être performantes. Les coûts d'une telle solution peuvent rapidement devenir importants car chaque client doit exécuter des traitements lourds.

  De plus, la charge réseau risque d'être importante car les traitements nécessitent des données qui ne seront peut-être pas présentées à l'utilisateur.

- Thin clients

  Contrairement au fat clients, des techniques tels que triggers, stored procedures et autres, permettant de placer la logique applicative sur le serveur vont être utilisées.
  Les clients peuvent alors fonctionner sur des configurations plus modestes. La surcharge réseau va aussi diminuer puisque seul les résultats destinés à l'utilisateur final seront transférés.

  Cette solution n'est toutefois pas dépourvue d'inconvénients.
  La logique applicative est implémentée sous forme de stored procedure. Le langage utilisé par celles-ci est propriétaire. De ce fait, on se lie fortement à un vendeur de base de données.
  Un autre inconvénient est que la logique se trouve dans la base de données. Dans le cas de systèmes distribués, le changement de la logique implique un changement sur l'ensemble des bases de données.

Il est communément accepté que ce modèle convient pour des systèmes comportant un maximum de 100 utilisateurs.

3. Les architectures Client/Serveur Three-Tier

Cette architecture apparue depuis peu va venir insérer un middle-tier entre le client et le serveur. Ce middle-tier va prendre en charge la logique applicative.

une architecture Three-Tier est composée :

- d'un client, en charge de la présentation des informations à l'utilisateur,
- d'un middle-tier, s'occupant de la logique applicative,
- d'un serveur de données (base de données, autres services, …)

Cette solution offre divers avantages tel que :

- **Séparation de la présentation, du traitement et de la sauvegarde des données.** Cette granularité plus fine permet de modifier un des composants sans que cela ne nécessite une adaptation des autres couches.
- **La surcharge réseau diminue**. Seules les données nécessaires sont transférées sur les clients.
  Bien-sûr, des données plus conséquentes doivent transférer entre le middle-tier et le ou les serveurs de données. Mais, dans ce cas, on se trouve dans un environnement de serveurs. Ces serveurs étant en moins grand nombre que les clients, on peut se permettre d'augmenter les ressources en terme de hardware, réseaux, …
- Une plus **grande indépendance vis à vis des bases de données**. Contrairement au modèle Two-Tier, la logique ne se trouve pas dans des stored procedure. Les accès aux données se font sur base du langage SQL qui lui, est plus ou moins standardisé.
- **Les langages utilisés pour le client et le middle-tier peuvent être différents**. Cela permet de tirer parti des atouts de chacun en fonction des besoins.
- **Amélioration de la qualité des solutions**. Le fait de scinder la présentation et le traitement entraîne une meilleure découpe des traitements.

C. La situation actuelle

La police judiciaire est présente sur tout le territoire belge. Elle est divisée de 27 brigades. Chaque brigade dispose d'un réseau local reliant l'ensemble des PC entre eux.
Ces réseaux locaux sont interconnectés entre eux par l'intermédiaire du réseau du ministère de la Justice. Toutefois, ce réseau est progressivement remplacé par celui de l'ex Gendarmerie.
A cela s'ajoute le site du Commissariat général de la Police Judiciaire.

Les données traitées, encodées ou consultées par les enquêteurs se trouvent à divers endroits :

- en brigade.
- au commissariat général.
- dans d'autres services de police.
- dans d'autres organisations tel que le Registre National, le ministère de la Justice, la direction pour l'immatriculation des véhicules
- …

Ces données se trouvent dans des bases de données diverses (Oracle, informix, sybase, access, …)

Le modèle de programmation principalement utilisé est le modèle Client/Serveur avec des clients lourds.
Les données en provenance du ministère de la Justice, de la gendarmerie et de certains autres services tiers sont accessibles par l'intermédiaire d'émulations terminales.

A l'heure actuelle, la consultation de toutes ces données (Procès-verbaux, données financières, …) nécessitent l'emploi de clients différents.

Les programmes (clients et serveurs) ont été développés dans des langages différents tel C, C++, SqlWindows, Centura, Delphi.
Le langage utilisé est fonction, de l'époque à laquelle les projets ont été réalisés et des programmeurs en charge du développement.

D. Les besoins

La police est une organisation qui traite énormément d'informations. Ces informations viennent de tous les horizons. Pour tirer profit de ces données, il faut les stocker et les mettre à la disposition d'un maximum de personnes. Donner ces données de façon brute n'est pas intéressant. Il faut les traiter, les analyser.
Cela implique, entre autre, de détecter des corrélations entre des données apparemment sans rapport, d'effectuer des statistiques pour mieux cibler les ressources nécessaires, …
Cette distribution de l'information doit être la plus transparente possible et ne doit pas être un frein à l'enquête. Un policier a d'autres choses à faire que de patienter derrière son ordinateur.

De plus, le parc informatique de la police n'est pas facilement renouvelable (budget). On se retrouve donc parfois avec des services disposant d'ordinateurs relativement dépassés. Toutefois, pour des raisons évidentes d'efficacité, les informations doivent être disponibles à tous les niveaux.

Mettre en place et maintenir un tel environnement demande énormément de ressources humaines. Il serait intéressant de pouvoir concentrer nos moyens, uniquement sur les business process.

Exemples de besoins.

Pour illustrer cette problématique, voici quelques exemples concrets de besoins du policier.

Le contrôle des personnes, véhicules,...

Dans toute enquête, interviennent des personnes, soit, en tant que préjudiciés, de suspects ou de témoins. La base d'une enquête est, dans un premier temps, de contrôler l'ensemble de ces personnes. Ce contrôle va servir à déterminer :

- L'identité exacte des personnes.
- Leur passé judiciaire
- Les liens éventuels qui relient certaines de ces personnes
- …

En ce qui concerne les objets ayant été employés dans un délit, la situation est similaire.

A l'heure actuelle, ces contrôles demandent énormément de ressources. En effet, les données sont disponibles mais esseminées dans diverses bases de données tel que le Registre National, la D.I.V. (Immatriculation des véhicules), les bases de données expertes, …

Chaque consultation nécessite l'emploi d'un programme différent. De par cet état de fait, les données ne sont ni comparées, ni recoupées entre elles automatiquement.

Pour des enquêtes de petites envergures, le problème est relativement simple.
Malheureusement, les enquêtes deviennent de plus en plus conséquentes. (Trafic international, mafia, …) Dans ce cas, le nombre de personnes et d'objets intervenants devient très élevé.
Le temps alors passé à effectuer ces contrôles augmente de façon exponentielle. Le nombre d'intervenants, pour une affaire importante, mais limitée à un environnement judiciaire, peut atteindre 4000.
Cela démontre le besoin énorme de la police en terme de traitement de l'information.

E. Solutions envisagées

D'emblée, une solution comportant un middle-tier semble s'imposer.
Ce type d'environnement apporte des solutions séduisantes :

- Un traitement est écrit une fois et est utilisable par l'ensemble des applications.
- Les traitements sont effectués sur des machines dédiées. Les stations clientes ne font que présenter les résultats.
- La maintenance et l'ajout de fonctionnalités sont simplifiés.

1. Les produits de middleware existants

Il existe une multitude de produits de ce genre. Toutefois, trois sortes du lot : CORBA, RMI et les Enterprise Java Beans.

a) CORBA

Corba (Common Object Request Broker Architecture) est une solution développée par l'OMG[1]. Ce consortium a été créé en 1989 pour promouvoir le développement et le déploiement d'applications distribuées dans des environnements hétérogènes.

Corba est une spécification basée sur l'OMA (Object Model Architecture). L'OMA définit l'architecture de base pour l'utilisation d'objets distribués dans des environnements hétérogènes.

-1- Object Model Architecture

L'OMA est composé de deux modèles. L'*object model* et le *reference model*.
Le premier définit comment décrire des objets capables d'être distribués dans des environnements hétérogènes. Le deuxième définit les interactions entre ces objets.

Les objets de l'OMA sont des entités fournissant des services. Ces services sont accessibles par l'intermédiaire d'**interfaces** bien définies. Une des caractéristiques principales de ces objets est que l'endroit où ils sont implémentés est caché, vis à vis du client. Le mécanisme mettant en place cette fonctionnalité s'appel l'ORB (Object Request Broker).

L'OMA définit plusieurs types d'interfaces :

1. **Object services** : Ces interfaces définissent l'accès à des services indépendants du domaine de l'application mais nécessaire au bon fonctionnement du système.
   Ex : Le *Naming Service*. Celui-ci permet aux clients de trouver des objets en se basant sur le nom de ceux-ci.

   L'OMG a définit des « Object service » dans un ensemble de domaines tel que la sécurité, la gestion des transactions, …

2. **Common Facilities** : Celles-ci définissent l'accès à des services indépendants de touts domaines d'applications mais qui sont orientés utilisateurs finaux.
   Ex : Le *Distibuted Document Component Facility (DDCF)*.

3. **Domain Interfaces** : Ces interfaces définissent des services spécialisés dans un domaine d'application particulier. Il existe des interfaces standardisées pour des domaines tel que les télécommunications, le monde médical, le monde des finances, …

[1] Object Management Group

4. **Application Interfaces** : Ces interfaces ne sont pas prédéfinies. Elles sont définies par les développeurs pour leurs applications propriétaires.

Une autre notion a été introduite dans le reference model. La notion d'*object framework*. Cela consiste à regrouper des objets d'un domaine particulier. Ces différents objets interagissent entre eux et sont accédés de l'extérieur par l'intermédiaire d'une seule interface. L'interface d'object framework. Cela permet de fournir des services évolués basés sur un ensemble d'objets simples.

-2- Common Object Request Broker Architecture

Corba est donc une spécification développée par l'OMG basée sur l'OMA.

La spécification CORBA 2.0 a été définie en 1995. Celle-ci comprend la spécification de :

- **l'ORB Core.**

  Ce service va acheminer les requêtes vers les objets et les réponses vers les clients et cela, de la manière la plus transparente possible.

  Les caractéristiques principales d'un ORB sont les suivantes :

  - L'utilisateur n'a pas à se soucier de l'endroit où se trouve les objets. L'ORB se charge de les retrouver.
  - La manière dont les objets ont étés implémentés (Langage, système d'exploitation, machine) est cachée.
  - L'ORB se charge d'activer automatiquement un objet dès lors qu'un de ses services est requis.
  - L'utilisateur ne sait pas quel mécanisme de communication est utilisé pour les transferts entre le client et les objets.
  - Fournit un annuaire des services.
    Celui-ci permet à un client de retrouver un objet sur base de son nom ou de ses propriétés. (Naming Service et Trading Service)

- **L'OMG Interface Definition Language (IDL)**

  Ce langage permet la définition des interfaces (Nom du type de l'objet, opérations et types supportés).
  L'utilisation de ce langage permet de séparer la définition de l'interface d'un objet, de son implémentation. Cette fonctionnalité est nécessaire pour permettre l'hétérogénéité voulue des environnements. (*pourquoi*)
  Les types utilisés dans ce langage devant être compris par toute une pléthore de systèmes, ils ont été standardisés.
  Les types définis sont les suivants :

  - Types de bases (long, short, float, …)
  - Types construits (Constructed types). Ces types sont équivalent aux **structures** du C et aux **Record** du Pascal.
  - Container (sequence, tableau, …)
  - Object Reference Types. Permet de stocker une référence vers un objet.

  Une autre caractéristique importante de ce langage, est de supporter l'héritage des interfaces. Une interface peut hériter d'une autre interface et lui adjoindre de nouvelles méthodes.

- **La correspondance des langages. (Language Mappings)**

  Les entités utilisées par Corba (interface, objets, opérations, types) doivent, à un certain moment être traduites vers un langage particulier sur une machine particulière. Cette spécification définit comment cette traduction doit être faite.
  Cette traduction est standardisée pour certains langages (C,C++,Smalltalk, Ada, COBOL, ...). Malheureusement, il n'existe pas de standardisation pour tous les langages. De ce fait, certaines implémentations de CORBA peuvent poser problème dans le cas d'une utilisation dans un environnement hétérogène.

- **Annuaire d'interfaces (interface Repository)**

  Pour qu'une application Corba utilise un objet, elle doit connaître son interface. Pour ce faire, il existe deux méthodes.
  La première consiste à intégrer l'interface lors de la compilation de l'application. Le désavantage de cette méthode tient au fait que si l'on change l'interface de l'objet, on risque de devoir recompiler l'application utilisant cet objet.
  La deuxième méthode fait usage d'un 'Interface Repository' (IR) pour découvrir l'interface lors de l'exécution. Cette méthode est surtout utilisée en conjonction avec l'invocation dynamique vue plus loin dans ce chapitre.

- **Stubs and Skeletons (Static invocation)**

  Les stubs et les skeletons sont deux morceaux de programmes générés par les outils Corba. Ces programmes vont se charger d'effectuer le « language mapping ». Le « stubs » crée les requêtes pour le client et le « skeleton » délivre la requête à l'objet. Le « stubs » fait donc partie intégrante de l'application cliente et le « skeleton » fait partie de l'objet. C'est pour cela que cette méthode est dite d'invocation statique.

- **Invocation dynamique (Dynamic Invocation and Dispatch)**

  Corba supporte deux interfaces pour l'invocation dynamique :

  Dynamic Invocation Interface (DII)

  Le DII permet à des clients d'utiliser des objets sans pour autant connaître son interface lors de la compilation. C'est une « Stub » générique qui va dès lors être utilisée.
  Le DII supporte plusieurs modes d'invocation des objets.

L'invocation synchrone : (Synchronous Invocation)	C 'est la méthode la plus courante. Le client effectue la demande et attend la réponse de l'objet.
L'invocation synchrone différée : (Deferred Synchronous Invocation)	Cette méthode permet à un client d'envoyer une réponse, d'effectuer d'autres tâches et de récupérer le résultat plus tard.
L'invocation à sens unique : (oneway Invocation)	Le client envoie une requête et n'attend aucune réponse de la part de l'objet.

  L'utilisation du DII est plus coûteuse en terme de ressource que l'invocation statique. Il faut en être conscient lors de son utilisation.

  Dynamic Skeleton Inteface (DSI)

  Cette méthode est la même que le DII, mais, au niveau du serveur. Le serveur peut alors utiliser des objets dont il ne connaissait pas l'interface lors de la compilation.

  Le DSi a été introduit dans la spécification du CORBA 2.0. Cela, pour permettre l'écriture de

passerelles entre ORB utilsant des protocoles de communication différents.

- **L'adaptateur d'objets (Object Adapters)**

  Ce composant joue le rôle d'intermédiaire entre l'implémentation d'un objet et l'ORB.

  Cet object adapter a en charge :

  - Génération et interprétation des « object references ».
  - Invocation des méthodes.
  - Sécurité des interactions.
  - Activation et désactivation des objets.
  - Correspondance entre des références d'objets et leurs implémentations.
  - Enregistrement de l'implémentation des objets.

  Le souhait de l'OMG est qu'il y ait peu d'implémentations de ces Object Adapters ; mais, que celles-ci soient largement diffusée de par le monde.
  Exemple d'Object Adapter : BOA (Basic Object Adapter) , POA (Portable Object Adapters).

- **Inter-ORB protocoles**

  Les spécifications du CORBA 1.0 ne définissaient pas le format des données transférées par les Obects Requests Broker. De ce fait, des ORB de vendeurs différents ne fonctionnaient pas entre eux.

  De ce fait, l'OMG a introduit ces spécifications dans CORBA 2.0

  Le GIOP (General Inter-ORB Protocol) définit le standard à utiliser pour des protocoles orientés connexion. Le IIOP (Internet Inter-ORB protocol), basé sur le GIOP spécifie le format pour des réseaux TCP/IP.

  Un autre standard, définit le format des données pour d'autres environnements (par exemple, ESIOP : Environement Specific Inter-ORB Protocol).

Le CORBA est un midle-tier particulièrement puissant. Toutefois, à mon sens, ses avantages sont aussi ses inconvénients. Ce système est tellement puissant qu'il en devient difficile à utiliser.

Avantages de CORBA

- Solution multi-environnements standardisée.
- Grande transparence de l'aspect distant des objets
- Invocation statique et dynamique

Inconvénient de CORBA

L'inconvénient majeur de Corba est la complexité de son environnement. C'est la contre-partie des avantages de celui-ci. La courbe d'apprentissage est, dès lors, relativement longue. Il faut, dans un premier temps, bien comprendre le rôle de chaque composant. Ensuite, il faut s'adapter aux types de données propres à Corba qui sont similaires mais différents des types originaux du langage de développement utilisé.
Un petit exemple pour démontrer cette complexité :
Lors de la définition d'une interface Corba, on trouve une multitude d'options que l'on peut spécifier pour en modifier son comportement. Ainsi, la définition d'une méthode peut être agrémentée de 4 options supplémentaires, et, pas moins de 12 options peuvent être spécifiées pour une propriété.

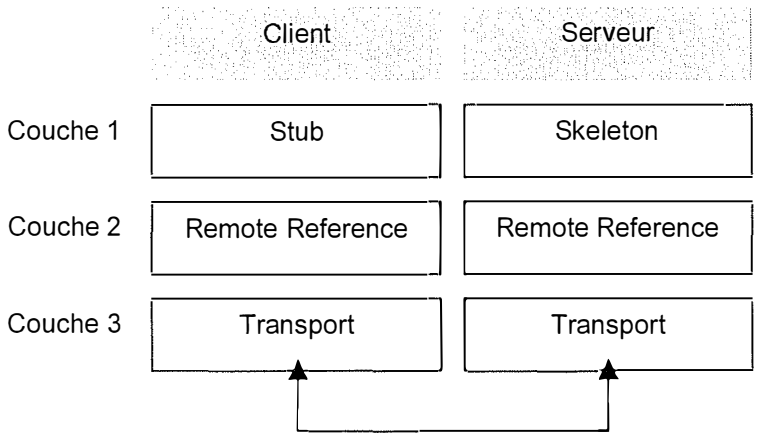
b) Le RMI (Remote Method Invocation)

Le RMI est la solution de middle-tier développée par SUN pour son environnement de développement fétiche, le JAVA.

Le RMI est beaucoup moins ambitieux que le CORBA. Mais, de ce fait, gagne en simplicité d'utilisation.

Contrairement à Corba, le RMI n'a pas de problème d'hétérogénéité des environnements. Le RMI ne fonctionne qu'en JAVA et, de plus, le code compilé en JAVA fonctionne sur tous les environnements grâce à l'utilisation d'une machine virtuelle.
Cela supprime totalement les problèmes de conversion de types rencontrés par Corba.
Ce langage étant relativement récent, il a été pensé pour l'utilisation de technologies comme le RMI.
Les classes peuvent être dérivées d'une classe de sérialisation permettant de transformer la classe en question en une suite d'octets. Le transfert d'arguments au travers d'un réseau ne pose donc aucun problème.

Le RMI est constitué de trois couches. Ces couches ont des tâches bien particulières selon qu'elles agissent en tant que client ou que serveur.



1. La couche 1 (Stub et Skeleton)

   Cette couche joue le rôle, comme dans Corba, d'interface entre le client et le système RMI en ce qui concerne le « stub » et entre le RMI et le serveur en ce qui concerne le « skeleton ».

   Cette couche transforme donc les requêtes et les réponses en suite d'octets. Cette suite sera transmise à la deuxième couche.

   Le stub

   Le stub est le morceau de programme compilé avec l'application cliente.

   Il s'occupe :

   - de transformer les arguments d'une méthode en suite d'octets.
   - d'initialiser l'appel à un objet remote.
     de transformer une suite d'octets en classes et types JAVA.
   - d'informer l'application cliente de la terminaison de l'appel

Le skeleton

Il est directement lié à l'objet serveur.

Il s'occupe :

- de transfomer la suite d'octets reçue en types et classes JAVA
- d'appeler l'objet désiré.
- de transformer la réponse en une suite d'octets

2. La couche 2 (Remote Reference)

Cette couche contient l'intelligence du système RMI.

Son rôle est :

- d'informer la couche transport du serveur sur lequel la requête doit être envoyée ou du client sur lequel la réponse doit être envoyée.
  Remarque : RMI supporte l'envoi d'une requête sur plusieurs serveurs. Le client prendra la réponse du premier qui lui répond
- de rétablir la communication en cas de rupture de celle-ci.

3. La couche 3 (Transport)

cette couche est responsable de la connexion physique avec l'autre partie.

Son rôle est :

- gérer les connexions
- écouter les arrivées de données, les demandes de connexion.
- maintenir une liste des objets distants créés.

Avantages du RMI

- Grande intégration dans l'environnement JAVA
- Les arguments passés peuvent être des objets.
- Simplicité d'utilisation

Inconvénients du RMI

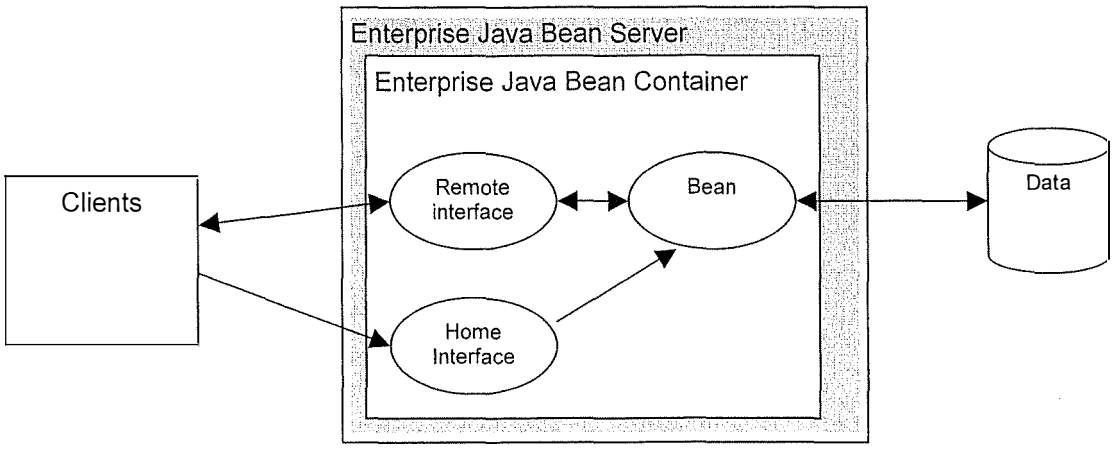
- Fortement lié à l'environnement JAVA
- Aucune transparence. Le client doit connaître l'endroit où réside l'objet.

Il existe aussi une autre technique appelée RMI sur IIOP (Internet Inter-Orb Protocol). Celle-ci permet alors de développer des clients et des objets Java fonctionnant dans un environnement CORBA et donc, de tirer parti du meilleur des deux mondes.

c) Les EJB (Enterprise Java Beans)

La spécification des EJB définit une architecture de composant pour permettre la construction d'application distribuée en Java.
Toutefois, l'architecture des EJB permet une intéraction entre des EJB et des langages autres que JAVA.

Les apports majeurs des EJB par rapport à d'autres middle-tier sont, les notions de transactions et de sécurités.
Un EJB est un objet RMI (parfois CORBA) **transactionnel** *et* **sécurisé**.



La « Home Interface » fournit les méthodes nécessaires à la localisation, la création et la destruction d'un « Bean ». Cette interface est maintenue par le « container » et est donc commune à l'ensemble des objets.

La « Remote Interface » est définie par un bean, qui, de cette manière, expose au monde extérieur ses méthodes.

Il existe deux types d '« enterprise beans » :

Session beans : Ces objets ressemblent aux objets Corba et RMI vus précédemment. Ils représentent un client au sein de l'EJB server pour la durée de sa connexion.

Entity beans : Ces objets sont nettement plus persistant que les sessions beans. Ils peuvent servir à plusieurs clients simultanément et ont une durée de vie qui dépasse celle des clients connectés. Ces objets sont souvent utilisés pour représenter des données en provenance de base de données.

Avantages des EJB

- EJB est en passe de devenir un standard d'entreprise pour le développement d'applications distribuées. Cette standardisation permet la composition d'objets provenant de différents horizons.
- L'aspect transactionnel est entièrement géré sans que le programmeur ne doive s'en soucier.
- Vu le langage sous-jacent (JAVA), les EJB héritent des avantages de celui-ci tel que le déploiement sur des plates-formes multiples sans recompilations nécessaires.

Inconvénients des EJB

- Les performances ne sont pas toujours au rendez-vous de par le langage utilisé.

2. Solution envisagée

Au vu des besoins de la Police, il ne fait aucun doute de l'utilité d'un middle-tier. Le nombre de clients est élevé, les machines clientes peu puissantes et les besoins en terme de traitements de l'information sont importants.

Une solution basée sur le système de SUN est directement écartée de par sa dépendance avec le langage JAVA. Le langage de référence de la Police étant maintenant le Delphi.
Pourtant, pour l'avoir déjà utilisé dans un projet, celui-ci est relativement séduisant. Il offre une grande simplicité de mise en œuvre et d'utilisation.

Que reprocher au Corba ? Il peut tout faire ou presque. Il est certain que faire ce choix ne serait pas contestable. Toutefois, il faut rester réaliste ; La police judiciaire, du moins pour l'instant, n'a pas les reins assez solides que pour supporter un tel système. Les ressources sont limitées tant en matériel qu'en personnel.

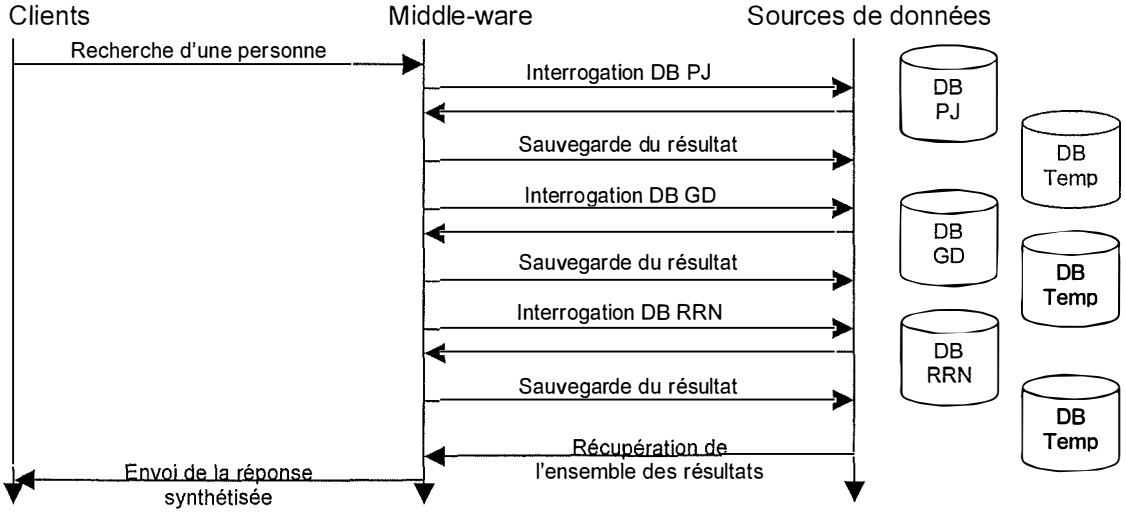
Une solution plus raisonable serait de mettre au point un système propriétaire qui intégrerait le meilleur de chacune des solutions précitées. Cela, sans s'encombrer de choses dont nous n'avons pas besoin.
Cette solution permettrait aussi d'ajouter des fonctionnalités à ce middle-tier. En effet, la plupart des middle-tier ne font que mettre à disposition des clients, des objets qui fonctionnent sur des machines distantes et ce, sans que le client s'en aperçoive. Un middle-tier propriétaire permettrait d'intégrer des services plus évolués dans le but d'encore simplifier la réalisation de systèmes distribués.

Prenons par exemple, la recherche par les policiers d'une entité (personnes, véhicules, ...) dans la base de connaissances des systèmes de la Police. Si nous voulons réduire cette recherche à sa plus simple expression, il nous faudra développer un objet capable d'interroger l'ensemble des bases de données de la police.
Cette recherche peut nous amener un certain nombre de réponses. Ces réponses, il faut les stocker afin de renvoyer une seule réponse synthétisée au policier.

Cela va donc nécessiter :

- de créer une ressource sur un serveur (vraisemblablement une table dans une base de données)
- de stocker l'information de manière temporaire
- de gérer les données contenues dans cette table. (Effacement après envoi de la réponse, ...)

Toutes ces tâches seront à faire dans chaque objet ayant besoin de ce type de service. Un système propriétaire permettrait d'intégrer cette fonctionnalité au sein même du middle-tier. Les objets n'auront alors plus qu'à utiliser ce service.



II. Spécifications

A. Les objectifs

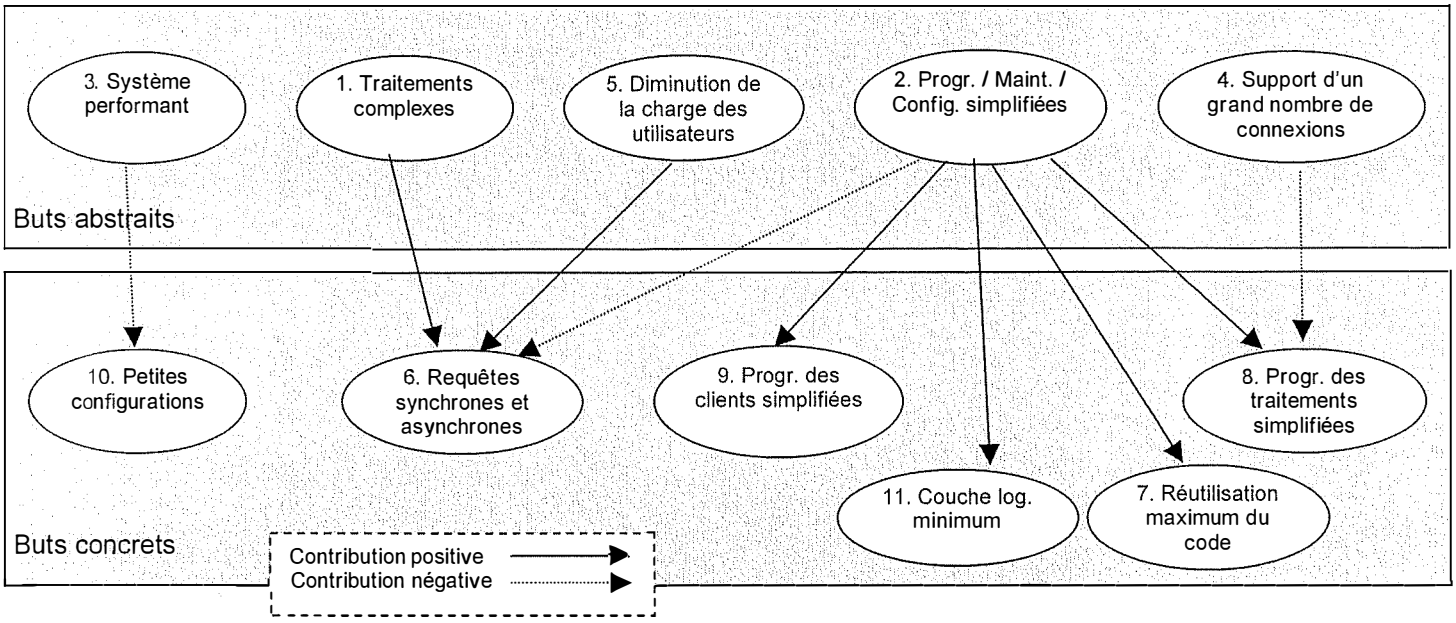
Au vu des besoins de la police et des choix faits au chapitre précédent, des objectifs vont pouvoir être déterminés.

1. Buts abstraits

1. Support de traitements complexes.
   -> Détection de corrélation entre données de systèmes différents.
2. Programmation / Maintenance / Configuration simplifiées.
   -> Concentrer les ressources sur les problèmes essentiels (Business process)
3. Système performant.
   -> Le système doit pouvoir effectuer des traitements complexes dans un temps acceptable.
4. Support d'un grand nombre de connexions.
   -> Amener l'information à un maximum de personnes.
5. Diminuer la charge des utilisateurs.
   -> Les policiers pourront alors passer plus de temps à l'enquête à proprement parler

De ces objectifs génériques, des buts plus concrets en terme de développement informatique, peuvent être dérivés.

2. Arborescence des buts.



3. Buts concrets

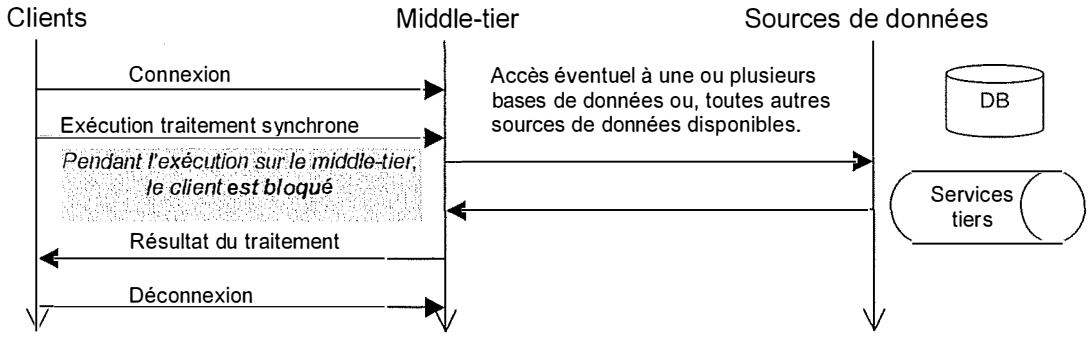
6. Support de requêtes synchrones, asynchrones.
7. Réutilisation maximum du code
8. Programmation des objets distants simples.
9. Programmation des clients simples.
10. Fonctionnement du client sur des petites configurations matérielles.
11. Limitation de la couche logiciel nécessaire au client.

B. Uses cases

Ces use cases ont été définis en fonction des exemples décrits à la page 11.

1. Scénario 1

Exécution par un client d'un traitement synchrone.



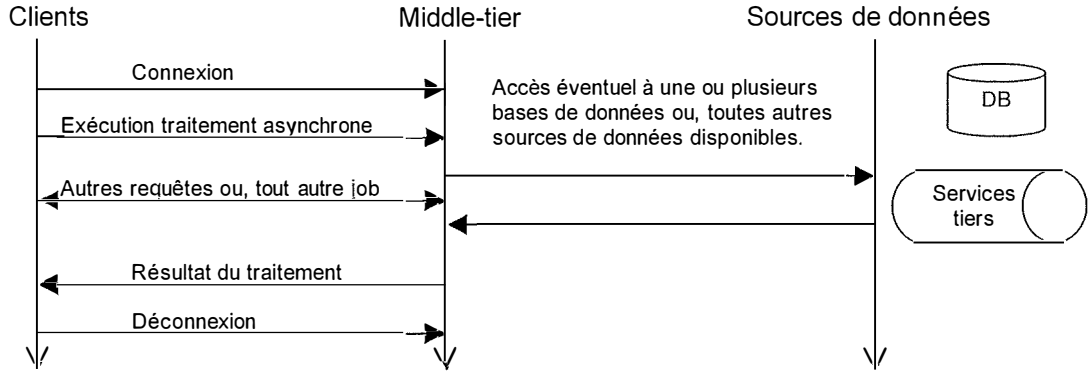
Remarque : L'appel synchrone est bloquant pour le client.

Ce type de schéma va permettre de répondre à des interrogations ponctuelles.

Ex : L'agent 22, peut ainsi facilement, consulter le registre national de monsieur durand pour vérifier son identité avant de commencer son audition.

2. Scénario 2

Exécution par un client d'un traitement asynchrone.



Remarque :

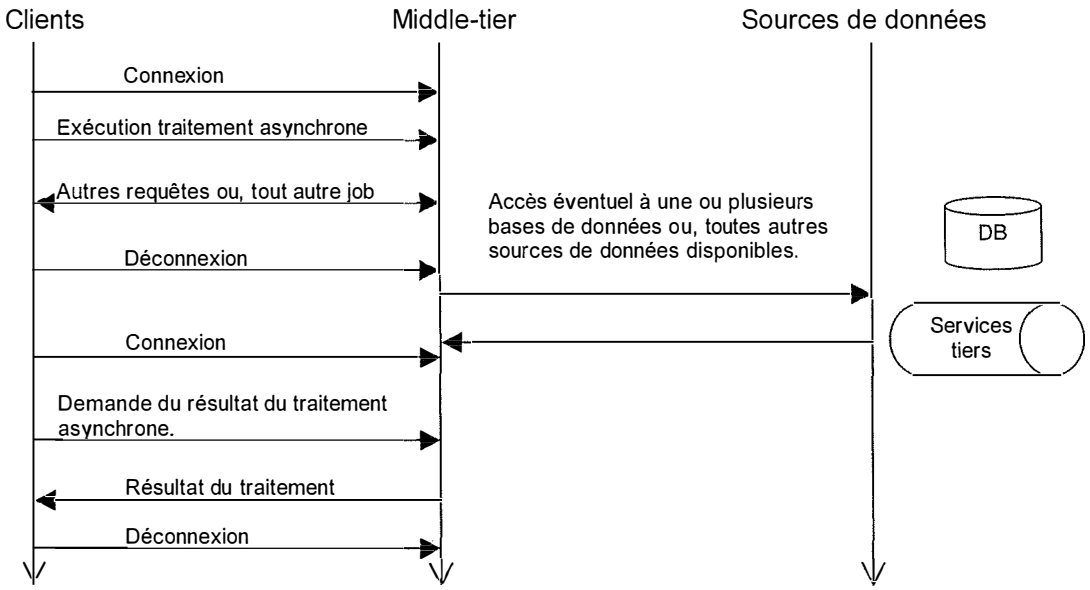
- Un appel asynchrone n'est pas bloquant pour le client.
- Durant une requête asynchrone, rien n'empêche l'appel à d'autres requêtes synchrones ou asynchrones.

Ce type de schéma va permettre l'exécution de requêtes plus complexes et donc plus longues à exécuter sans pour autant bloquer le client. Il aura le loisir de continuer à utiliser son application sans attendre le résultat de cette requête.

Ex : Le chef de section de l'agent 22, lui demande les statistiques de l'année précédente en ce qui concerne les vols à main armée et les vols avec violence. L'agent 22 lance le programme de statistiques, envoie la première requête et, immédiatement après, la deuxième requête. Notre agent peut maintenant continuer la rédaction de son PV sans que le calcul des statistiques ne lui occasionne le moindre désagrément.
Une fois son PV terminé, il vérifiera si les statistiques sont disponibles.

3. Scénario 3

Exécution par un client d'un traitement asynchrone long (Répartition de l'exécution d'une même requête sur plusieurs connexions d'un même client)



Remarque :
Un appel asynchrone long, permet de demander l'exécution d'un traitement et de récupérer le résultat plus tard, même après une déconnexion physique du client.

Ce type de schéma pourra être très utile pour des traitements longs.

On peut prévoir, dans le cadre de l'exemple précité page 11, la mise au point d'un traitement effectuant la consultation de l'ensemble des bases de données de connaissances, et ce, pour une liste de personnes ou d'objets. Cela déchargerait considérablement le policier.

L'avantage de ne pas être lié à une connexion physique est de pouvoir se préserver des plantages de la machine et de libérer totalement la machine client. L'on pourra ainsi éteindre la machine, la redémarrer dans un autre environnement ou, tout simplement, ne pas être encombré par une application qui fonctionne depuis plusieurs jours.

La problématique de la récupération des réponses sera abordée ultérieurement.

Ex : Notre agent 22 a en charge une affaire de meurtre. Après enquête, il se retrouve avec une liste de témoins. Une des premières choses à faire, est de vérifier, au sein des bases de données policières, si l'on connaît quelque chose sur l'un d'eux.
L'agent 22 va donc envoyer au système une requête avec, comme paramètre, la liste des témoins. Après cela, il est 18 heures, notre agent n'a qu'une seule envie, rentrer chez lui. Il éteint son ordinateur comme demandé par la note de service n°32458. Le lendemain matin, il allume son ordinateur, lance son programme de consultation et demande au serveur s'il a la réponse à la requête envoyée la veille. Le serveur ayant eut la nuit entière pour traiter la demande, celui-ci, renvoie les informations que la police dispose sur les témoins.

C. Les contraintes

1. Contraintes techniques

Les contraintes du projet sont les suivantes :

- Le protocole réseau supporté doit être TCP/IP.
- Le langage de développement sera Delphi 5.

L'avènement de la police unique a donné lieu à une standardisation. Le langage de référence est le Delphi 5 et le protocole réseau, le TCP/IP.

2. Contraintes non fonctionnelles

Les temps de réponse du serveur d'applications seront corrects. Toutefois, il ne s'agit pas d'un système temps réel.
Pour tenter d'optimiser les performances et ainsi d'améliorer le service rendu aux clients, on introduira un concept de priorité d'exécution pour les requêtes asynchrones. Cela permettra d'assigner une faible priorité aux tâches lourdes et non urgentes.

Le middle-tier doit supporter un certain nombre d'utilisateurs. Certains services fonctionnent avec plus de 250 personnes. Le système doit pouvoir les supporters. Bien-sûr, la machine sur laquelle fonctionne le middle-tier devra être dimensionnée en fonction.

Le middle-tier doit pouvoir fonctionner en 24x7. Les services de police fonctionnant 24 heures sur 24, il est normal de fournir un service continu.

Le concept de transaction sera géré au niveau des business objects. En aucun cas, le middle-tier ne gérera des transactions. La gestion des transactions par le middle-tier est un processus complexe qui, à mon sens, n'est pas nécessaire dans notre contexte.

Le middle-tier devra implémenter une sécurité au niveau des accès au business objects, sur base d'un identifiant et d'un mot de passe.
L'accès de ces utilisateurs pourra éventuellement être limité à une partie du réseau (Sous réseau en TCP/IP) toutefois, ceci n'est pas une priorité.

Une sécurisation au niveau du transfert des données n'est pas nécessaire. Des solutions globales pour les réseaux de police sont mises en place.

III. Analyse fonctionnelle

A. Lignes de force du projet

La liste des buts concrets déterminés dans le chapitre précédent va permettre de faire ressortir les grandes lignes du projet.

1. Le principe de la signature unique.

Le produit auquel on veut arriver doit être accessible à tout un chacun. (Programmation simplifiée à tous les niveaux)
De par mon expérience dans le domaine des middle-tiers[1], il me semble que la définition de l'interface et sa maintenance est quelque chose de relativement compliqué.
En effet, si l'on prend un produit comme CORBA, il faut définir une interface par l'intermédiaire d'un outil particulier sous Delphi.
Pour chaque méthode ou propriété, il faut déterminer le type Corba correspondant le mieux aux besoins. La liste des types proposée en comporte près de 70 différents !
Si le nom de certain sont sans équivoque, ce n'est pas le cas pour tous. (Ex : BStr, IString *, LPStr, ...)
Chaque définition de méthode ou de propriété peut être agrémentée par des « modificateurs » ou des « flag »
Au niveau des valeurs de retour des méthodes, tous les types ne sont pas acceptés.

Une fois l'interface définie, un fichier est généré. Ce fichier doit être utilisé lors de la compilation du client et du serveur.
cette interface doit donc être maintenue, parallèlement à la définition de l'objet.

Cette relative « lourdeur » a toutefois l'avantage de rendre l'utilisation d'objets distants presque transparente lors du développement d'applications distribuées.

Néanmoins, Il est possible d'imaginer une solution permettant d'améliorer cette procédure sans perdre trop de fonctionnalités.

Il suffit de définir des objets dont **toutes les méthodes auraient la même interface**.
Chaque méthode prendrait pour paramètre un **objet évolué** et renverrait un autre objet évolué.
Cet objet évolué pourrait contenir l'ensemble des paramètres nécessaire au bon déroulement de la méthode.

Avantages

- On supprime ainsi la nécessité de définir l'interface ou, du moins, on la réduit à sa plus simple expression. (Le nom de la méthode)
- Cet objet pourrait contenir un nombre indéfini de variables. Cet aspect ouvre des portes intéressantes car, un appel à une méthode pourrait renvoyer directement l'ensemble des lignes d'une table de code dans un seul objet.

Inconvénients

- L'appel d'une méthode ressemblera un peu moins à un appel traditionnel car, il faudra préparer cet objet, y incorporer les variables et seulement après, faire l'appel à la méthode voulue.
- L'utilisation d'un fichier définissant l'interface permet une documentation automatique des paramètres acceptés par les méthodes. Dans la solution proposée, il sera impératif de maintenir une documentation séparée des paramètres acceptés.

[1] Mon expérience se limite au développement d'une application en RMI et à l'étude réalisée dans le cadre de ce mémoire.

Les avantages qu'offre cette approche, semblent intéressants par rapport aux désagréments que cela occasionne.

2. Objets d'applications simples

Les objets d'application sont les entités fonctionnant sur le middle-tier qui implémentent les traitements propres aux applications distribuées.
Bien que ces objets doivent pouvoir traiter des problèmes complexes, il faut tenter de simplifier au maximum leur conception et leur utilisation.

C'est pourquoi, ces objets d'applications doivent :

- Pouvoir être testés facilement, en local, comme s'il s'agissait d'objets normaux.
- Pouvoir être regroupés en librairie. Cette librairie sera alors traitée par le serveur d'application.
- Pouvoir servir pour une exécution synchrone ou asynchrone (même long) indifféremment sans que cela ne nécessite le moindre changement au niveau du code de l'objet.

3. Le concept de ResultSet

La plupart des implémentations voit le middle-tier comme une extension du client. On exécute un objet distant comme s'il se trouvait en local. L'avantage de cette façon de faire est de pouvoir concentrer les ressources nécessaires sur un petit nombre de machines (middle-tier), ainsi que de faciliter la maintenance du code, …
Cette façon de cacher l'aspect distant de l'objet est une bonne chose ; toutefois, il est possible d'améliorer le service rendu aux clients.

Regardons du coté des bases de données et surtout, du mode de programmation utilisé actuellement pour interagir avec celles-ci.
Lorsque l'on veut récupérer un ensemble de lignes d'une table, on exécute une query. Cette query retourne un resultset. (Sorte de pointeur sur le résultat du query qui se trouve sur le serveur).
L'ensemble des lignes n'est pas transféré en une fois sur le client. Chaque ligne du résultat est transférée une à une à la demande du client.

Il serait intéressant d'intégrer cette fonctionnalité au middle-tier. Cela permettrait d'écrire facilement des objets d'application renvoyant une grosse quantité de données, sans pour autant, devoir les rapatrier sur le client en une seule fois. Ces données pourraient alors êtres « découpées » en plusieurs entités logiques transférées sur le client séparément.

Je propose donc d'introduire dans le serveur d'application le concept de resultset. Ce service permettra, à n'importe quelle méthode d'un objet, de créer un resultset qui sera accessible par le client sans que l'objet d'application aie à s'en préoccuper. Il faut évidemment conserver les deux modes de fonctionnement. Une méthode d'un objet doit aussi pouvoir renvoyer un résultat « traditionnel » .

Dans le cadre de l'exemple du scénario 3 page 22, chaque témoin sur lequel on a une information pourrait faire l'objet d'une ligne du resultset. Ainsi, l'agent 22 pourra facilement parcourir les données sans délais, même si le volume total des informations est conséquent. (Puisque l'accès à ces données se fera témoin par témoin)

4. Maître mot : Dynamique

Le serveur d'application doit être simple d'utilisation tout en assurant une qualité du service.

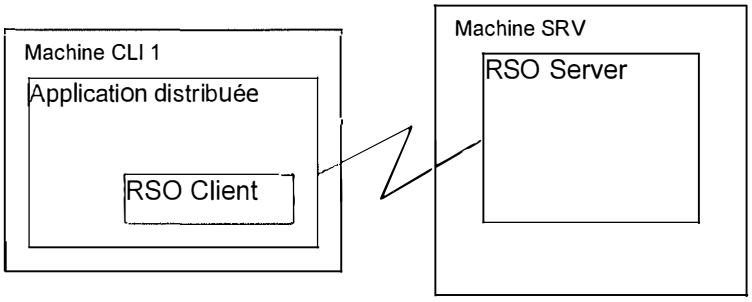
Cela implique donc un ensemble de choses tel que :

- Ajout et retrait de librairies d'O.A. simples
- Fonctionnement du serveur d'application en 24x7

Le serveur d'application doit donc être prévu pour pouvoir enregistrer et dés-enregistrer des librairies d'O.A. dynamiquement sans que cela n'influence le fonctionnement de celui-ci.

B. Architecture globale du système

Le terme RSO est un terme générique pour identifier le projet. (Remote Server Object).
Le principe est de faire fonctionner le middle-tier (RSO Server) sur une machine dédicacée à cet effet. (Machine Srv)
Tout client exécutant une application distribuée se connectera au RSO server par l'intermédiaire d'une couche logicielle spécifique. (RSO Client). Le RSO Client utilisera le protocole TCP pour effectuer cette connexion.



C. Identification des divers composants

Les différents modules à développer sont :

Modules	Description
RSO Client	Encapsule l'ensemble des fonctionnalités nécessaire à l'exécution des requêtes par un client.
RSO Serveur	Encapsule l'ensemble des fonctionnalités nécessaire au traitement des requêtes par le serveur d'application.

Pour mener à bien ce développement, il est nécessaire d'identifier les différentes tâches que devront effectuer ces deux modules et ensuite de tenter de regrouper ces tâches en unités indépendantes pouvant être développées séparément. J'appellerai chacune de ces unités des services.
Ces services seront utilisés, pour certains, uniquement par le serveur et pour les autres, par le serveur et le client.

Premièrement, nos deux modules doivent pouvoir communiquer. Il faut donc prévoir un *service réseau* qui fournira les outils nécessaires pour acheminer les informations d'une machine à une autre.

Bien que cela n'ait pas encore été déterminé, le RSO server sera vraisemblablement composé soit, de plusieurs threads, soit de plusieurs processus. Il est impératif que des données puissent être échangées entre ces processus. Un *service de communication inter-process* sera donc prévu.

Pour mettre en œuvre le principe de la signature unique évoqué à la page 24, un outil implémentant un objet évolué pouvant contenir un ensemble de données de types différents est nécessaire.
De plus, il pourra servir de base pour transformer cet ensemble de données en un seul bloc de données. Ce bloc sera alors facilement acheminable au travers du réseau. Et bien sûr, le service pourra décomposer ce bloc en un ensemble de données équivalent aux données initiales.
Ce service s'appellera *service sérialisation*.

Au sein du RSO Server, il faudra pouvoir exécuter les traitements contenus dans les objets d'applications. Ce sera la charge du *service objet d'application*.

Le concept de resultset introduit page 25 demande des traitements particuliers. Le serveur devra maintenir un ensemble de resultsets généré par les méthodes des objets d'applications. Cette fonctionnalité sera fournie par le *service Resultset*.

Le faite de supporter les requêtes asynchrones nous obligent à mettre en place en un service dédicacé à cette tâche. Lorsque le serveur termine l'exécution de sa requête, il doit garder trace du résultat de celle-ci pour pouvoir l'envoyer au client lorsqu'il le demandera.
Ce service s'appellera *AsynchRequest*.

Chaque utilisateur connecté au serveur devra disposer d'un contexte sur celui-ci. C'est à dire une zone qui gardera en mémoire ce que l'utilisateur a créé comme objet d'application, quels sont les Resulset auquel il a accès, etc ... Le service en charge de maintenir ce contexte par utilisateurs connectés s'appellera *Session*.

La sécurité sera implémentée par le service *sécurité*. Ce service maintiendra une liste des utilisateurs autorisés à ce connecté au middle-tier en question ainsi que les objets d'applications qu'il a le droit d'utiliser.

Enfin, de par mon expérience en programmation sous Delphi, je sais qu'il manque certains outils pour gérer des agrégats de données sous forme de liste, ...
Ce service contiendra donc les outils de gestion de données manquant au Delphi.
Ce service s'appellera *service structuration de données*.

Résumé des services à développer :

ID	Services	Fonction
SSD	Structuration de données	Fournit des outils de base pour la gestion de données tel que des listes,...
SNT	Réseau	Gère les communications réseaux.
SIC	IPC	Gère les communications inter-processus.
SSR	Sérialisation	Gère la sérialisation et la dé-sérialisation d'un ensemble de données afin de pouvoir les acheminer à travers le réseau par paquets.
SOA	Objet d'application	Gère l'exécution des objets d'applications.
SSC	Sécurité	Gère les utilisateurs et des droits d'accès au serveur d'applications.
SST	ResultSet	Gère les ResultSet
ASY	AsynchRequest	Gère les requêtes asynchrones.
SSM	Session	Gère la liste des sessions courantes du serveur d'application.

D. Protocole d'échange entre le client et le serveur.

Nous allons décrire ici le protocole qui sera supporté par le client et le serveur. Celui-ci devra permettre aussi bien l'envoi de commandes avec des paramètres que le renvoi de résultats du serveur aux clients.

Ce protocole est basé sur le client. Le serveur ne prendra jamais l'initiative de dialoguer avec un client. Le serveur répondra toujours à une requête du client. Cette méthode permet de simplifier au maximum le client. Chaque demande du client impliquera d'office une réponse du serveur. Même dans le cas de requêtes asynchrones, le serveur répondra par une sorte d'acquiescement. Cela permet de toujours attendre une réponse du serveur avant de rendre la main à l'utilisateur et de ne pas avoir à garder une trace du contexte d'exécution.

Comme décrit dans les uses cases, un objet d'application peut avoir une durée de vie plus longue que la connexion de l'utilisateur. (Transaction longue) Il nous faut donc définir un terme plus générique pour décrire la période de validité de l'ensemble des objets d'applications créés par un utilisateur. La notion de **session** sera alors utilisée.

Cette session débutera lors de la connexion d'un utilisateur. Elle se terminera, lorsque celui-ci le notifiera. Toutefois, si l'utilisateur veut laisser une session active, il devra se détacher proprement de la session par une commande appropriée. Cela, pour permettre au serveur de différencier une déconnexion normale, d'un « plantage » de la machine client.

Attention : Un utilisateur pourra, bien entendu, créer plusieurs sessions simultanées. Chaque message envoyé au serveur comportera un identifiant de session. Cet identifiant n'est pas strictement nécessaire vu le protocole utilisé (Une requête envoyée au client nécessite toujours une réponse du serveur) toutefois, il est intéressant de le prévoir pour des évolutions futures.
Dans un souci de clarté, **cette information n'est pas reprise dans la description de chaque message du protocole**.

Figure III-1 - Liste des messages du protocole.

Message	Description	Sens
SESSION_CONNECT	Création ou attachement à une session	Client -> Serveur
SESSION_DISCONNECT	Déconnexion ou destruction d'une session	Client -> Serveur
OA_CREATE	Création d'un objet d'application	Client -> Serveur
OA_FREE	Destruction d'un objet d'application	Client -> Serveur
OA_EXEC	Exécution synchrone d'une méthode d'un objet d'application	Client -> Serveur
OA_EXEC_ASYNCH	Exécution asynchrone d'une méthode d'un objet d'application	Client -> Serveur
OA_EXEC_ASYNCH_RESULT	Récupération du résultat de l'exécution asynchrone d'une méthode	Client -> Serveur
RESULTSET_FIRST	Positionnement au début d'un ResultSet	Client -> Serveur
RESULTSET_LAST	Positionnement à la fin d'un ResultSet	Client -> Serveur
RESULTSET_NEXT	Positionnement à l'élément suivant d'un ResultSet	Client -> Serveur
RESULTSET_PREV	Positionnement à l'élément précédent d'un ResultSet	Client -> Serveur
RESULTSET_GET	Récupération de l'élément courant d'un ResultSet	Client -> Serveur
RESULTSET_COUNT	Récupération du nombre d'éléments présents dans un ResultSet	Client -> Serveur
RESULTSET_FREE	Destruction d'un ResultSet	Client -> Serveur

A chacun de ces messages sont associés 2 autres messages (MESSAGE_OK, MESSAGE_KO) qui sont toujours renvoyés par le serveur au client.

a) Connexion d'un client.

Demande de connexion d'un client au serveur.

Requête

SESSION_CONNECT	UTILISATEUR	MOT DE PASSE	ID SESSION

Utilisateur / Mot de passe : Identifiant de l'utilisateur. Cet utilisateur doit être connu du système.
ID Session : Identifiant d'une session existante. Si ce paramètre est nul, une nouvelle session doit être créée.

Réponses

SESSION_CONNECT_OK	ID SESSION

SESSION_CONNECT_KO	CODE ERREUR	DETAIL

Code Erreur : Code de l'erreur survenue pour faciliter l'automatisation du traitement de l'erreur.
Détail : Raison détaillée du refus de la connexion sous forme textuelle. Ce champ existe malgré le code d'erreur précédent pour donner un maximum de détails sur la raison de l'échec.

Erreurs possibles
ID Session invalide
Utilisateur ou mot de passe invalide

b) Déconnexion / détachement d'un client

Demande au serveur de terminer définitivement ou temporairement une session.

Requête

SESSION_DISCONNECT	ID SESSION	MODE DE DECONNEXION

ID Session : Identifiant de la session à déconnecter
Mode de déconnexion : Indique si la session doit être définitivement détruite, ou si elle doit rester active pour permettre une re-connexion ultérieure de l'utilisateur.
Les différents modes de déconnexion sont repris à la Figure III-2 : Mode de déconnexion

DISCONNECT :	Demande de déconnexion normale. La requête est refusée s'il existe encore des sessions actives.
DISCONNECT FORCE :	Demande de déconnexion forcée. Dans le cas où une ou plusieurs sessions existent, elles sont détruites. Les objets qui y sont associés sont aussi détruits.
DETACH :	Demande de déconnexion tout en laissant la ou les sessions actives.

Figure III-2 : Mode de déconnexion

Réponses

DISCONNECT_OK

DISCONNECT_KO	CODE_ERREUR	DETAIL

Code Erreur : Code de l'erreur survenue.
Détail : Raison détaillée de l'échec.

Erreurs possibles
ID Session invalide
Mode de déconnexion invalide

c) Création d'un objet d'application

Requête

OA_CREATE	NOM LIBRAIRIE	NOM OBJET	PARAMETRE CONSTRUCTEUR OBJET

Nom Librairie : Nom de la librairie dans laquelle se trouve l'objet voulu.
Nom Objet : Nom de l'objet dont on veut créer une instance
Paramètre Constructeur Objet : Paramètre du constructeur de l'objet.
Attention, ce paramètre est un objet évolué contenant l'ensemble des paramètres que l'on désire passer au constructeur de l'objet.

Réponses

OA_CREATE_OK	ID OA

ID OA Identifiant de l'objet d'application remote.

OA_CREATE_KO	CODE ERREUR	DETAIL

Code Erreur : Code de l'erreur survenue
Détail : Détail de l'erreur

Erreurs possibles
ID Session invalide
Nom de librairie invalide
Nom d'objet invalide

d) Destruction d'un objet d'application

Requête

OA_FREE	ID OA

ID OA Identifiant de l'objet d'application remote à détruire.

Réponses

OA_FREE_OK

OA_FREE_KO	CODE ERREUR	DETAIL

Code Erreur : Code de l'erreur survenue
Détail : Détail de l'erreur

Erreurs possibles
ID Session invalide
ID Objet d'application invalide

e) Exécution d'une méthode d'un objet d'application de façon synchrone

Requête

OA_EXEC	ID OA	NOM METHODE	PARAMETRE METHODE

ID OA : Identifiant de l'objet d'application remote dont on veut exécuter une méthode.
Nom Méthode : Nom de la méthode à exécuter
Paramètre Méthode : Paramètre à passer à la méthode.
Attention, ce paramètre est un objet évolué contenant l'ensemble des paramètres que l'on désire passer à la méthode.

Réponses

OA_EXEC_OK	ID RESULTSET	RESULTAT

ID ResultSet : Identifiant du ResultSet éventuel.
Résultat : Résultat de la méthode.

OA_EXEC_KO	CODE ERREUR	DETAIL

Code Erreur : Cf. ci-dessus.
Détail : Cf. ci-dessus.

Erreurs possibles
ID Session invalide
ID OA invalide
Méthode inexistante

f) Exécution d'une méthode d'un objet d'application de façon asynchrone

Requête

OA_EXEC_ASYNCH	ID OA	NOM METHODE	PARAMETRE EXECUTION	PARAMETRE METHODE

ID OA : Identifiant de l'objet d'application remote dont on veut exécuter une méthode.
Nom Méthode : Nom de la méthode à exécuter
Priorité d'exécution : Priorité d'exécution de la méthode. (CF. **Figure III-3**)
Paramètres Méthode : Paramètre à passer à la méthode.

- HAUTE
- NORMALE
- BASSE

Figure III-3 - Priorité d'exécution d'une méthode

Réponses

OA_EXEC_ASYNCH_OK	ID REQUETE ASYNCH	ID RESULTSET

ID Requête Asynch : Identifiant de la requête asynchrone.
ID ResultSet : Identifiant du ResultSet éventuel.

Remarque : Par rapport à une exécution synchrone, aucun résultat n'est retourné. Ce qui est logique vu que la méthode n'a pas encore été exécutée. Une fois celle-ci exécutée, le résultat pourra être récupéré grâce au message « OA_EXEC_ASYNCH_RESULT » détaillé plus loin.
Par contre, un identifiant de la liste de résultats est retourné, même si cette liste est encore vide. Cet identifiant servira plus tard pour récupérer ces résultats par l'intermédiaire de la fonction « RESULTSET_GET »

OA_EXEC_ASYNCH_KO	CODE ERREUR	DETAIL

Code Erreur : Cf. ci-dessus.
Détail : Cf. ci-dessus.

Erreurs possibles
ID Session invalide
ID OA invalide
Méthode inexistante

g) Résultat d'une requête asynchrone.

Requête

OA_EXEC_ASYNCH_RESULT	ID REQUETE ASYNCH

ID Requête Asynch : Identifiant de la requête asynchrone.

Réponses

OA_EXEC_ASYNCH_RESULT_OK	RESULTAT

Resultat : Résultat de la requête. Si la requête n'a pas encore été exécutée ou est en cours d'exécution, le résultat est nul.

OA_EXEC_ASYNCH_RESULT_KO	CODE ERREUR	DETAIL

Code Erreur : Cf. ci-dessus.
Détail : Cf. ci-dessus.

Erreurs possibles
ID Session invalide
ID Requête asynchrone invalide

h) Positionnement au début d'un ResultSet

Requête

RESULTSET_FIRST	ID RESULTSET

ID ResultSet : Identifiant du ResultSet.

Réponses

RESULTSET_FIRST_OK	EOF	DONNEES

EOF : ResultSet Vide ?
DONNEES : Données extraites du ResultSet

Remarque : Pour une question d'optimisation, les données courantes du ResultSet sont incluses à la réponse générée par les fonctions de positionnement dans le ResultSet.

RESULTSET_FIRST_KO	CODE ERREUR	DETAIL

Code Erreur : Cf. ci-dessus.
Détail : Cf. ci-dessus.

Erreurs possibles
ID Session invalide
ID ResultSet invalide

i) Positionnement à la fin d'un ResultSet

Requête

RESULTSET_LAST	ID RESULTSET

ID ResultSet : Identifiant du ResultSet.

Réponses

RESULTSET_LAST_OK	EOF	DONNEES

EOF : ResultSetVide ?
DONNEES : Données extraites du ResultSet

RESULTSET_LAST_KO	CODE ERREUR	DETAIL

Code Erreur : Cf. ci-dessus.
Détail : Cf. ci-dessus.

Erreurs possibles
ID Session invalide
ID ResultSet invalide

j) Positionnement d'un ResultSet à l'élément suivant

Requête

RESULTSET_NEXT	ID RESULTSET

ID ResultSet : Identifiant du ResultSet.

Réponses

RESULTSET_NEXT_OK	EOF	DONNEES

EOF : Fin du ResultSet ?
DONNEES : Données extraites du ResultSet

RESULTSET_NEXT_KO	CODE ERREUR	DETAIL

Code Erreur : Cf. ci-dessus.
Détail : Cf. ci-dessus.

Erreurs possibles
ID Session invalide
ID ResultSet invalide

k) Positionnement d'un ResultSet à l'élément précédent

Requête

RESULTSET_PREV	ID RESULTSET

ID ResultSet : Identifiant du ResultSet.

Réponses

RESULTSET_PREV_OK	EOF	DONNEES

EOF : Début du ResultSet ?
DONNEES : Données extraites du ResultSet

RESULTSET_PREV_KO	CODE ERREUR	DETAIL

Code Erreur : Cf. ci-dessus.
Détail : Cf. ci-dessus.

Erreurs possibles
ID Session invalide
ID ResultSet invalide

l) Récupération de l'élément courant d'un ResultSet

Requête

RESULTSET_GET	ID RESULTSET

ID ResultSet : Identifiant du ResultSet.

Réponses

RESULTSET_GET_OK	DONNEES

DONNEES : Données extraites du ResultSet

RESULTSET_GET_KO	CODE ERREUR	DETAIL

Code Erreur : Cf. ci-dessus.
Détail : Cf. ci-dessus.

Erreurs possibles
ID Session invalide
ID ResultSet invalide

m) Nombre d'élément présent dans le ResultSet

Requête

RESULTSET_COUNT	ID RESULTSET

ID ResultSet : Identifiant du ResultSet.

Réponses

RESULTSET_COUNT_OK	NOMBRE ELEMENT

Nombre Elément : Nombre d'élément du ResultSet.

RESULTSET_COUNT_KO	CODE ERREUR	DETAIL

Code Erreur : Cf. ci-dessus.
Détail : Cf. ci-dessus.

Erreurs possibles
ID Session invalide
ID ResultSet invalide

n) Supprime un ResultSet

Requête

RESULTSET_FREE	ID RESULTSET

ID ResultSet : Identifiant du ResultSet.

Réponses

RESULTSET_FREE_OK

RESULTSET_FREE_KO	CODE ERREUR	DETAIL

Code Erreur : Cf. ci-dessus.
Détail : Cf. ci-dessus.

Erreurs possibles
ID Session invalide
ID ResultSet invalide

E. Analyse des services

1. Service structuration de données (SSD)

Les différents services auront besoin de garder des informations en mémoire et de pouvoir les retrouver facilement. Ce service rassemblera l'ensemble des outils nécessaires à cette tâche.

a) Outil Liste.

La liste est le moyen le plus approprié pour maintenir une relation entre éléments de même nature mais n'ayant rien à voir les uns avec les autres. (Ex. : Liste d'utilisateurs)

Ce type d'agrégat sera fortement utilisé par les autres services. (Ex : Liste des utilisateurs, liste des sessions, ...)

Bien que ce type d'agrégat existe au sein de Delphi, il est intéressant de l'encapsuler pour pouvoir lui adjoindre facilement de nouvelles fonctionnalités.

Les éléments d'une liste seront accessibles soit, par un numéro, soit par un identifiant de type texte.
Le numéro de suite unique est fourni par le système lors de l'insertion dans la liste.
L'identifiant texte est fourni au système lors de l'insertion dans la liste. Cet identifiant n'est pas obligatoirement unique.
Une donnée de type et de longueur variable peut être associée à chaque élément de la liste.
Des méthodes doivent exister pour permettre de retrouver un élément sur base de son numéro ou de son identifiant.

Primitives de base de l'outil Liste :

Primitives	Descriptions
Create	Création d'une liste. *Paramètres* : – / *Retour* : – Identifiant liste
Free	Destruction d'une liste. *Paramètres* : – Identifiant liste *Retour* : – /
Add	Ajout d'un élément dans la liste *Paramètres* : – Identifiant liste – Identifiant texte – Données *Retour* : – Identifiant élément (Numéro de suite)

Insert	Insertion d'un élément dans la liste. Attention, une insertion va générer un décalage dans les numéros de suite. *Paramètres* : – Identifiant liste – Position de l'insertion (Numéro de suite) – Identifiant texte. – Données *Retour* : – Identifiant élément (Numéro de suite)
Delete	Effacement d'un élément dans la liste sur base du numéro de suite ou de l'identifiant texte. Si celui-ci n'est pas unique, tous les éléments ayant cet identifiant seront supprimés. *Paramètres* : – Identifiant liste – Identifiant texte ou numéro de suite. *Retour* : –
Get	Récupération d'un élément de la liste sur base de son numéro de suite. *Paramètres* : – Identifiant liste – Numéro de suite. *Retour* : – Données
IndexOf	Récupération du premier élément de la liste ayant pour identifiant celui passé en paramètre. *Paramètres* : – Identifiant liste – Identifiant texte. *Retour* : – Numéro de suite.
Count	Récupération du nombre d'élément présent dans la liste. *Paramètres* : – Identifiant liste *Retour* : – Nombre d'éléments

2. Service Réseau (SNT)

Ce service va fournir les outils de base pour gérer l'ensemble des communications réseau aussi bien pour les clients que pour le serveur.

Bien que les fonctionnalités nécessaires soient différentes pour un client et pour un serveur, une partie de celles-ci est toutefois pareille.

Sans entrer dans des détails techniques pour l'heure, on entend par événement, l'arrivée de quelque chose en provenance du réseau (Connexion, données, déconnexion)

a) Client :

Support mono processus : Le service doit pouvoir s'adapter à des situations simples où l'on se trouve vraisemblablement dans un environnement mono processus.

Support multiple connexions : Plusieurs connexions à des serveurs distincts ou non, doivent pouvoir être gérées.

Liste des connexions : Une liste des connexions doit être maintenue permettant de retrouver les informations pertinentes de l'ensemble des connexions.
Les données minimumes à conserver sont reprises à la Figure III-4

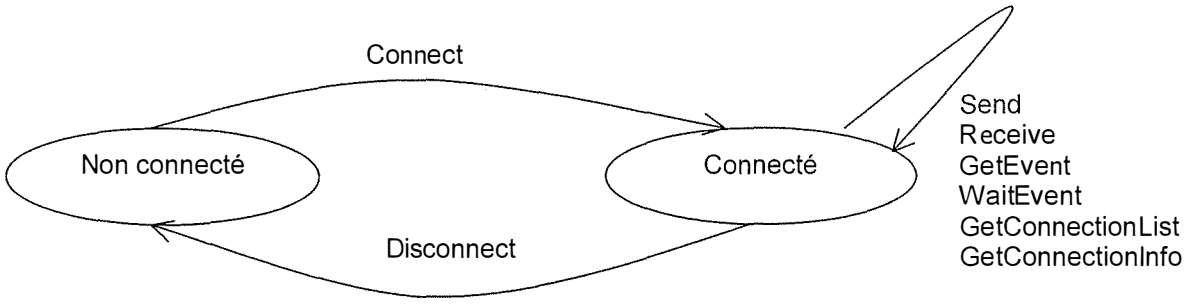
Primitives de base du composant réseau client:

Primitives	Descriptions
connect	Envoie une demande de connexion. *Paramètres* : – Adresse du serveur *Retour* : – Identifiant connexion
disconnect	Envoi une demande de déconnexion. *Paramètres* : – Identifiant connexion. *Retour* : – Résultat disconnect
send	Envoie des données *Paramètres* : – Identifiant connexion – Données à envoyer *Retour* : – Résultat send
receive	Reçoit des données *Paramètres* : – Identifiant connexion *Retour* : – Données
waitEvent	Attend l'arrivée d'un événement sur une connexion *Paramètres* : – Identifiant connexion – Timeout *Retour* : – Event reçu ou erreur si Timeout

GetConnectionList	Renvoi une liste des connexions actuelles. *Paramètres* : – / *Retour* : – Liste des connexions

Diagramme d'état du composant réseau client :

Ce diagramme d'état représente une connexion à un serveur du point de vue d'un client.



b) Serveur :

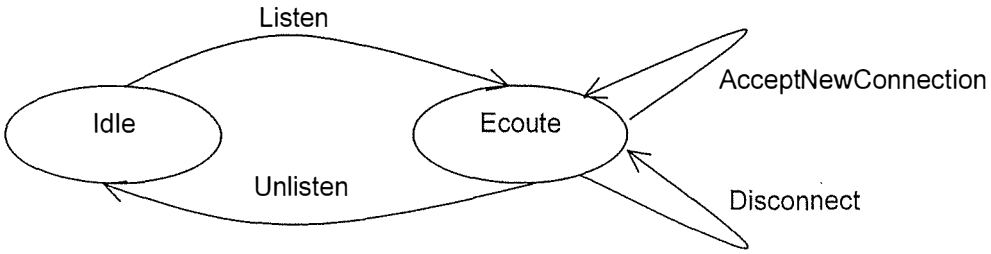
Support multi processus :	Le service doit aussi être capable de s'intégrer dans un environnement complexe (gestion des communications gérée par plusieurs processus) Le traitement d'une même connexion par différents processus doit donc être possible.
Support multiple connexions :	Le serveur doit, bien-sûr, supporter plusieurs connexions de clients distincts ou non. Par contre, le nombre de processus nécessaires ne doit pas être fonction du nombre de connexion.
Liste des connexions :	Une liste des connexions établies par les clients doit être maintenue permettant de retrouver les informations pertinentes de l'ensemble de ces connexions. Les données minimums à conserver sont reprise à la Figure III-4, page 41

Primitives de base du composant réseau serveur :

Primitives	Descriptions
Listen	Ecoute l'arrivée de nouvelle connexion. *Paramètres* : – Adresse d'écoute *Retour* : – /
Unlisten	Stoppe l'écoute de nouvelle connexion. *Paramètres* : – Adresse d'écoute *Retour* : – /
AcceptNewConnection	Accepte ou refuse une nouvelle connexion. *Paramètres* : – / *Retour* : – Identifiant connexion client
disconnect	Envoi une demande de déconnexion. *Paramètres* : – Identifiant connexion client *Retour* : – /
Send	Envoie des données *Paramètres* : – Identifiant connexion client – Données *Retour* : – /
receive	Reçoit des données *Paramètres* : – Identifiant connexion client *Retour* : – Données
getEvent	Récupère les évènements associés à une connexion. *Paramètres* : – / *Retour* : – Identification connexion client
waitEvent	Attend l'arrivée d'un événement sur une connexion *Paramètres* : – / *Retour* : – /
GetConnectionList	Renvoi une liste des connexions actuelles. *Paramètres* : – / *Retour* : – Liste des connexions actives
GetConnectionInfo	Renvoie les informations d'une connexion *Paramètres* : – Identifiant connexion client *Retour* : – Information connexion (CF : Figure III-4 - information connexion)

Diagramme d'état du composant réseau serveur :

Ce diagramme d'état représente Les différents états possibles pour un serveur.



Remarques : Les autres primitives n'ont de raison d'être que lorsque la connexion est établie. Le diagramme d'état vu pour un client est alors d'application.

- Adresse réseau locale
- Adresse réseau distante
- Date et heure de la connexion

Figure III-4 - information connexion

3. Service IPC (SIC)

Ce service va fournir les fonctionnalités de base pour la communication inter processus.

a) Listes FIFO

Ces communications se feront sur base de listes FIFO (Premier entré, premier sorti) améliorées.

Les listes FIFO se prêtent bien au sujet qui nous intéresse. Le serveur aura à réagir à des requêtes venant de clients. Celles-ci arriveront dans un certain ordre et seront traitées dans le même ordre.

Les spécifications nous amènent à ajouter des fonctionnalités à ces listes :

1. Elles doivent supporter un aspect de priorité afin de pouvoir servir certaines requêtes plus rapidement que d'autre. La place de l'insertion dans la liste sera fonction de la priorité de l 'élément à insérer. Pour une même priorité, les règles d'usage des listes FIFO sont d'application.

2. Elles doivent supporter le multi-processus. (essence même d'un IPC)
   Cette fonctionnalité permettra à plusieurs processus de s'échanger des informations.
   Cet échange d'informations devra se faire de façon efficace. C'est pourquoi, une fonction de synchronisation devra permettre à un processus de « dormir » tant que la liste est vide.
   Le service devra utiliser un objet de synchronisation du système d'exploitation pour obtenir la solution la plus efficiente.

A une liste, sera associé un identifiant. Celui-ci permettra à un processus de « s'attacher » à une liste existante.
Toutefois, selon le système d'exploitation envisagé, il est raisonnable d'envisager une simplification de cette méthode.
Si le système permet l'exécution de plusieurs processus dans un même espace d'adressage (Thread sous Windows), on peut supprimer l'identifiant. Ce sera alors la même instance de la liste qui sera utilisée par l'ensemble des processus.

Primitives de base des listes FIFO :

Primitives	Descriptions
CreateFifoList	Crée une liste FIFO. *Paramètres* : – / *Retour* : – Identifiant liste FIFO
AttachFifoList	S'attache à une liste existante sur base d'un identifiant. *Paramètres* : – Identifiant liste FIFO *Retour* : – /
DetachFifoList	Se détache d'une liste. *Paramètres* : – Identifiant liste FIFO *Retour* : – /

DeleteFifoList	Détruit une liste précédemment créée. *Paramètres* : – Identifiant liste FIFO *Retour* : – /
PushFifoList	Place un élément dans une liste. *Paramètres* : – Identifiant liste FIFO – Données – Priorité *Retour* : – /
PopFifoList	Récupère un élément dans une liste. *Paramètres* : – Identifiant liste FIFO *Retour* : – Données
WaitPopFifoList	Attend un élément dans la liste et le renvoie *Paramètres* : – Identifiant liste FIFO – Timeout *Retour* : – Données

4. Service Sérialisation (SSR)

Ce service va permettre de stocker un ensemble de variables de types divers dans des « container ». Ces variables seront accessibles de façon distincte et aisée. Le service devra pouvoir générer un paquet de données contenant ces variables. Il sera aussi possible de faire l'opération inverse. C'est à dire de récupérer la valeur de chaque variable de façon distincte.

Ce service servira de base pour le stockage des paramètres à échanger entre un client et le serveur d'application. Il est la clé de la méthode de la signature unique.

Ce service sera capable de :

1. Stocker, modifier, effacer, énumérer des variables de tout type. (Chaîne, entier, booléen, date, flottant, suite d'octets, …)

2. Récupérer la valeur de chaque variable nominément pour la stocker dans un type de base du langage.

3. Générer et lire une suite d'octets représentant l'ensemble des variables. Cela permettra de transmettre facilement par le réseau l'ensemble des paramètres d'une méthode.

4. Le service devra pouvoir effectuer ces opérations de façon efficace. C'est à dire que l'ajout, le retrait ou la modification d'une variable ne doit pas déclencher un processus complexe de mise à jour d'une zone mémoire.

Primitives de base du service de sérialisation:

Primitives	Descriptions
CreateContainer	Créer un nouveau container *Paramètres* : – / *Retour* : – Identifiant container
DeleteContainer	Effacer un container *Paramètres* : – Identifiant container *Retour* : – /
SetVariable	Stocker une variable. *Paramètres* : – Identifiant container – Nom de la variable – Valeur de la variable *Retour* : – /
DeleteVariable	Effacer une variable. *Paramètres* : – Identifiant container – Nom de la variable *Retour* : – /

GetVariableList	Liste des variables stockées. *Paramètres* : – Identifiant container *Retour* : – Liste des noms de variables
GetVariable	Récupérer la valeur d'une variable. *Paramètres* : – Identifiant container – Nom de la variable *Retour* : – Valeur de la variable
SaveToStream	Générer une suite d'octets. *Paramètres* : – Identifiant container *Retour* : – Suite d'octets
ReadFromStream	Lire une suite d'octets. *Paramètres* : – Identifiant container – Liste d'octets *Retour* : – /

5. Service ResultSet (SST)

Ce service va permettre de stocker des informations sous la forme d'une table où chaque ligne sera accessible séparément. Celui-ci va centraliser la gestion de ces tables(ResultSet).

La sauvegarde de ces données se fera en mémoire pour des raisons de performance. Toutefois, par la suite, un mécanisme de transfert sur disque pourra être mis en place. Ce mécanisme permettrait alors de :

- Récupérer la mémoire vive occupée par les ResultSet trop gourmand.
- Réaliser une sauvegarde permanente de l'ensemble des ResultSet lors de la fermeture du serveur.

Chaque ligne du ResultSet sera représentée par un container en provenance du service sérialisation. Les avantages de l'utilisation de cette technique sont multiples :

- L'implémentation du concept de colonne est implicite.
- Grande souplesse du ResultSet, car le nombre de ces colonnes est variable, au sein d'un même ResultSet.
- Développement plus aisé, en utilisant des concepts identiques.

La contrepartie de cela est que chaque ligne contiendra les informations sur le nom des colonnes. (= Perte de place en mémoire)

Les fonctionnalités de base nécessaire sont les suivantes :

- Ajout d'une ligne de données au ResultSet
- Récupération d'une ligne de données
- Navigation dans le ResultSet

Primitives de base du service ResultSet:

Primitives	Descriptions
CreateResultSet	Création d'un nouveau ResultSet *Paramètres* : – / *Retour* : – Identifiant ResultSet
DeleteResultSet	Suppression d'un ResultSet *Paramètres* : – Identifiant ResultSet *Retour* : – /
SaveRow	Ajoute au ResultSet une ligne de données. *Paramètres* : – Identifiant ResultSet – Identifiant container *Retour* : – /
GetRow	Retourne le container contenant les données de la ligne courante *Paramètres* : – Identifiant ResultSet *Retour* : – Identifiant container

First, Last, Next, Prev	Permet la navigation au sein d'un ResultSet. *Paramètres* : – Identifian ResultSet *Retour* : – /
CountRow	Renvoi le nombre d'éléments présent dans un ResultSet *Paramètres* : – Identifiant ResultSet *Retour* : – Nombre d'éléments

6. Service AsynchRequest (ASY)

Ce service sera en charge de centraliser les informations nécessaires au bon fonctionnement des méthodes exécutées de manière asynchrone.

Cela consiste en la sauvegarde de :

- l'identifiant de la requête asynchrone.
- la requête en provenance du client
- l'identifiant du ResultSet.
- la réponse de la méthode.
- l'état de l'exécution.

Primitives de bases du service AsynchRequest :

Primitives	Descriptions
CreateAsynchRequest	Création d'une nouvelle requête asynchrone *Paramètres* : – / *Retour* : – Identifiant AsyncRequest
DeleteAsynchRequest	Supprime une requête asynchrone *Paramètres* : – Identifiant AsynchRequest *Retour* : – /
SetRequest	Sauvegarde la requête en provenance du client. *Paramètres* : – Identifiant AsynchRequest – Requête *Retour* : – /
GetRequest	Récupère la requête en provenance du client. *Paramètres* : – Identifiant AsynchRequest *Retour* : – Requête
SetReturnValue	Sauvegarde la valeur de retour de la méthode. *Paramètres* : – Identifiant AsynchRequest – Valeur de retour *Retour* : – /
GetReturnValue	Renvoi la valeur de retour de la méthode. *Paramètres* : – Identifiant AsynchRequest *Retour* : – Réponse
SetState	Change l'état d'une requête *Paramètres* : – Identifiant AsynchRequest – Etat *Retour* : – /

SetResultSetID	Sauvegarde l'identifiant du ResultSet associé. *Paramètres* : – Identifiant AsynchRequest – Identifiant ResultSet *Retour* : – /
GetResultSetID	Retourne l'identifiant du ResultSet associé. *Paramètres* : – Identifiant AsynchRequest *Retour* : – Identifiant ResultSet
SetErrorCode	Spécifie le code d'erreur généré lors de l'exécution de la méthode. *Paramètres* : – Identifiant AsynchRequest – Code d'erreur *Retour* : – /
GetErrorCode	Retourne le code d'erreur généré lors de l'exécution de la méthode. *Paramètres* : – Identifiant AsynchRequest *Retour* : – Code d'erreur
SetErrorDetail	Spécifie le message d'erreur généré lors de l'exécution de la méthode. *Paramètres* : – Identifiant AsynchRequest – Code d'erreur *Retour* : – /
GetErrorDetail	Retourne le message d'erreur généré lors de l'exécution de la méthode. *Paramètres* : – Identifiant AsynchRequest *Retour* : – Code d'erreur

7. Service Objet d'application (SOA)

a) Description :

Ce service sera en charge de la gestion et de l'exécution des objets d'applications.
Les objets d'applications(classes) seront regroupés en librairie d'O.A..
Chaque méthode des O.A. aura le même canvas. Elles prendront en paramètre un container du service de sérialisation et renverront en retour un autre container du service de sérialisation.
Les O.A. seront décrits plus en détails ultérieurement.

Ce service sera capable de :

- Enregistrer, désenregistrer des modules de manière dynamique sans re compilation.
- Sauvegarder les modules enregistrés de façon permanente.
- Créer / détruire une instance d'un objet d'application.
  Lors de la création, le service renverra alors, un identifiant d'instance. Celui-ci sera nécessaire pour l'exécution d'une méthode de cet objet et pour la destruction de celle-ci.
- Exécution d'une méthode d'un objet d'application donné avec passage à celui ci des paramètres fournit sous forme d'un service de sérialisation et renvois des paramètres de retour.

En aucun cas, ce service ne doit garder trace des instances existantes dans le système. Cette fonction sera attribuée au service de gestion des sessions. Les raisons de ceci seront expliquées lors de la description détaillée du service de gestion des sessions.

Primitives de base du service objet d'application:

Primitives	Descriptions
RegisterModule	Enregistrement d'un module. *Paramètres* : – Nom du module *Retour* : – /
UnregisterModule	Dés enregistrement d'un module. *Paramètres* : – Nom du module *Retour* : – /
SaveConfig	Sauvegarde permanente de la liste des modules enregistrés. *Paramètres* : – / *Retour* : – /
LoadConfig	Chargement de la liste des modules enregistrés. *Paramètres* : – / *Retour* : – /
CreateInstance	Création d'une instance d'un objet d'application dont le nom est passé en paramètre et renvoi de l'identifiant de l'instance. *Paramètres* : – Nom du module – Nom de l'objet d'application – Paramètre de création de l'objet. *Retour* :

	– Identifiant de l'instance
FreeInstance	Suppression d'une instance d'un objet d'application. *Paramètres* : – Identifiant de l'instance *Retour* : – /
ExecuteMethod	Exécution d'une méthode dont le nom et l'identifiant de l'instance sont passés en paramètre. *Paramètres* : – Identifiant instance – Nom de la méthode – Paramètre de la méthode *Retour* : – Valeur de retour de la méthode – Identifiant du resultset éventuelle

Remarque : Le traitement de variables de classe n'est pas prévu. Il pourra être envisagé par la suite.

La gestion distante de ces variables obligerait soit :

- à compliquer le pré-traitement de la compilation en générant des fonctions (Set & Read) pour traiter ces variables. Or, nous avons volontairement simplifié ce processus par l'introduction de la méthode de la signature unique.
- à utiliser des « astuces » du compilateur. Cette façon de faire rend le code beaucoup moins portable mais est très efficace et est simple à mettre en œuvre. Cette solution n'est donc pas écartée mais mise sur le coté dans un premier temps.

b) Interface de programmation des objets d'applications

L'un des buts, définit au début de ce travail, est de faciliter le développement d'objet d'application.

Pour rappel, un objet d'application est une collection de routines. (Typiquement, une classe)
Un ensemble d'OA forment une librairie d'OA. Cette librairie sera chargée par le serveur et l'ensemble des OA présents dans celle-ci seront utilisables par les clients qui en ont le droit.

Chaque méthode d'un OA aura la même signature a l'exception du constructeur de l'OA :
Méthode (Paramètre : SSR) : SSR
Constructeur (Paramètre : SSR)

La partie technique devra analyser une solution donnant le moins de surcharge possible aux programmeurs pour rendre ces OA accessibles par le serveur. Il n'est pas possible de donner une solution maintenant car celle-ci dépendra du type d'implémentation des librairies d'OA.

8. Service Sécurité (SSC)

Ce service s'occupera de la gestion des utilisateurs ainsi que les droits d'accès de ceux-ci.
Chaque utilisateur se connectant au serveur d'application devra au préalable avoir été enregistré par le service sécurité.
Ce service maintiendra une liste des utilisateurs valides reprenant les informations spécifiées à la Figure III-5

- Un identifiant.
- Un mot de passe.
- Son nom en clair.
- Un descriptif sous forme de texte libre.
- Une liste des modules d'objets d'applications auxquels il a accès.
- Un état : actif ou non actif. (si non actif, l'utilisateur ne peut se connecter)
- L'heure et la date de sa dernière connexion.

Figure III-5 Liste des attributs d'un utilisateur

La liste des utilisateurs sera stockée dans une liste du service Container. Les données seront stockées de façon permanente grâce au service de stockage.
L'accès aux informations d'un utilisateur se fera sur base de son identifiant.

Le **niveau de priorité** permettra de définir des classes d'utilisateurs. Certains utilisateurs auront donc toujours une priorité plus grande que d'autre.

Primitives de base du service sécurité :

Primitives	Descriptions
AddUser	Ajout d'un utilisateur. *Paramètres* : – Attributs d'un utilisateur *Retour* : – /
UpdateUser	Modification d'un utilisateur. *Paramètres* : – Attributs d'un utilisateur *Retour* : – /
DeleteUser	Effacement d'un utilisateur. *Paramètres* : – Identifiant utilisateur *Retour* : – /
GetUser	Récupération des informations d'un utilisateur. *Paramètres* : – Identifiant utilisateur *Retour* : – Attributs utilisateur

GetUserList	Renvois de la liste des identifiants utilisateurs. *Paramètres* : – / *Retour* : – Liste des identifiants utilisateurs
GrantAcces	Ajoute un accès à un module d'O.A. à un utilisateur *Paramètres* : – Identifiant utilisateur – Nom d'un module *Retour* : – /
RevokeAcces	Retire un accès à un utilisateur. *Paramètres* : – Identifiant utilisateur – Nom d'un module *Retour* : – /
SaveUsers	Sauvegarde permanente de la liste des utilisateurs. *Paramètres* : – / *Retour* : – /
ReadUsers	Lecture de la liste des utilisateurs sauvegardée précédemment par SaveList. *Paramètres* : – / *Retour* : – /

9. Service Session Manager (SSM)

Description :

Ce service est un élément clé du système. Celui-ci, va tenir à jour, l'ensemble des sessions courantes. Par courantes, on entend les sessions créées par les utilisateurs actuellement connectés ainsi que les sessions longues qui ont été créées mais pas encore fermées définitivement. Et cela, même si l'utilisateur n'est plus connecté.

Cette liste de session sera en fait le point d'entrée pour récupérer toutes les informations d'une session. (Ex. : liste des instances d'objets d'applications créés, utilisateur associé, …)

Chaque entrée dans cette liste contiendra les informations suivantes :

- Identifiant de session
- Type de session (Synchrone / Asynchrone)
- Utilisateur associé
- Liste des objets d'applications crée pour cette session (Par liste, on entend une méthode permettant d'y accéder)
- Date de création de la session

Figure III-6 - Liste des attributs d'une entrée de la liste du Session Manager

Ce service donnera la possibilité de créer / modifier / supprimer des sessions.
Lors de la création d'une session, une attribution automatique d'un numéro de session aura lieu. Ce numéro sera renvoyé au service créateur pour une utilisation ultérieure.

Ce service gère donc la liste des instances créées par une session. Il est préférable de faire comme cela plutôt que donner cette tâche au service de gestion des O.A. pour des raisons de sécurité.
En effet, en faisant de la sorte, on interdira à une session d'utiliser une instance d'objet qui aurait été créée par une autre session. Cela aurait aussi été possible dans le service précité mais, avec une plus grande complexité.

Primitives de base du SessionManager:

Primitives	Descriptions
CreateSession	Création d'une session. L'ensemble des paramètres nécessaire à cette création est alors passé. Un identifiant de session est renvoyé. *Paramètres* : – Attributs d'une session *Retour* : – Identifiant session
DeleteSession	Effacement d'une session donnée. *Paramètres* : – Identifiant session *Retour* : – /
GetSessionInfo	Renvoi des informations d'une session *Paramètres* : – Identifiant session *Retour* : – Attributs session

AddOA	Ajout d'un objet d'application à la liste d'OA d'une session *Paramètres* : – Identifiant session – Identifiant instance OA *Retour* : – /
DeleteOA	Retire un objet d'application à la liste des OA d'une session *Paramètres* : – Identifiant session – Identifiant instance OA *Retour* : – /
GetListOA	Renvoi la liste des OA pour une session donnée. *Paramètres* : – Identifiant session *Retour* : – Liste des identifiants d'instance d'OA
ExistOA	Teste l'existence d'un OA pour une session donnée. *Paramètres* : – Identifiant session – Identifiant instance OA *Retour* : – Booléen

F. Analyse des modules

1. Module Client (RSO CLI)

Description :

Ce service est en quelque sorte, le chef d'orchestre de la partie client du serveur d'application. C'est lui qui va mettre en œuvre les différents services nécessaires au bon fonctionnement du client.

Ses tâches principales sont :

1. Etablir et maintenir une connexion réseau avec le serveur d'application.
2. Traduire les méthodes appelées par le programmeur en message compréhensible par le serveur d'application. Le protocole est bien entendu celui vu au chapitre III.D page 28.
3. Envoyer ces messages au serveur d'application.
4. Récupérer les messages reçus du serveur d'application.
5. Traduire ces messages reçus et présenter les résultats au programme appelant sous une forme compréhensible par celui-ci.

Le service sérialisation sera utilisé dès qu'il s'agit de transférer des données du client vers le serveur et vice-versa. Ce service a été pensé dans ce but. Il permet de traiter facilement un ensemble de données (=variables) en une seule entité et de transformer cette entité en une suite de bytes facilement transférable.

Primitives de base du module client :

Ces primitives sont, bien entendu, calquées sur le protocole.

Primitives	Descriptions
SessionConnect	Création ou attachement à une session *Paramètres* : – Identification réseau du serveur – Identifiant éventuel d'une session précédente – Identification utilisateur *Retour* : – Identification session
SessionDisconnect	Déconnexion ou destruction d'une session *Paramètres* : – Identification session – Mode de déconnexion (CF.Figure III-2 page 29) *Retour* : – /
OACreate	Création d'un objet d'application *Paramètres* : – Identification session – Nom de la librairie – Nom de l'objet à créer – Paramètre éventuel à passer au constructeur de l'objet *Retour* : – Identification de l'instance d'OA créée.
OAFree	Destruction d'un objet d'application *Paramètres* : – Identification session – Identification de l'OA. *Retour* :

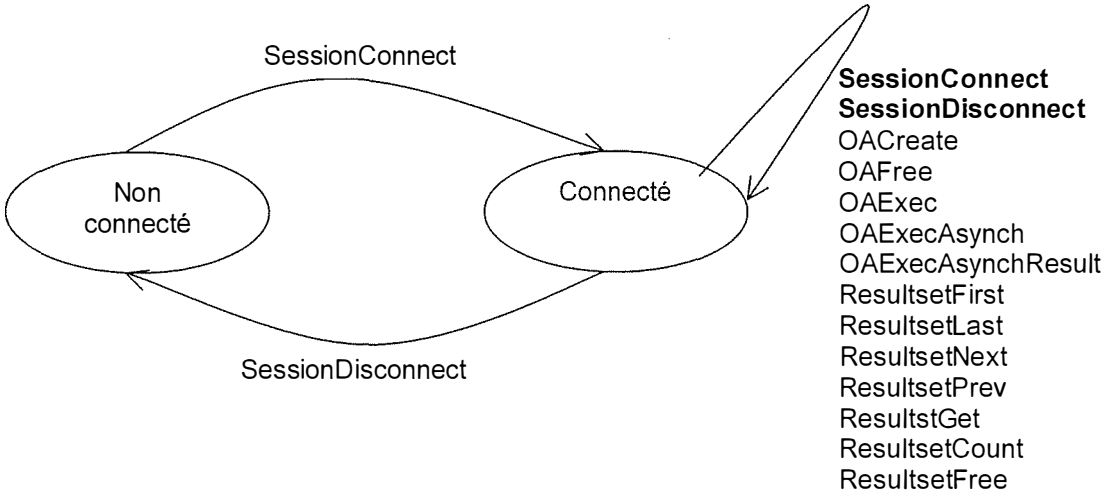
	– /
OAExec	Exécution synchrone d'une méthode d'un objet d'application *Paramètres* : – Identification session – Identification de l'OA. – Nom de la méthode à exécuter – Paramètres de cette méthode – Priorité d'exécution. (CF. **Figure III-3** Page 31) *Retour* : – Identifiant Resultset. – Résultats de la méthode
OAExecAsynch	Exécution asynchrone d'une méthode d'un objet d'application *Paramètres* : – Identification session – Identification de l'OA. – Nom de la méthode à exécuter – Paramètres de cette méthode – Priorité d'exécution. (CF. **Figure III-3** Page 31) *Retour* : – Identifiant de la requête asynchrone – Identifiant Resultset.
OAExecAsynchResult	Récupération du résultat de l'exécution asynchrone d'une méthode *Paramètres* : – Identification session – Identification de la requête asynchrone. *Retour* : – Résultats de la requête
ResultsetFirst	Positionnement au début d'un ResultSet *Paramètres* : – Identification session – Identification ResultSet. *Retour* : – /
ResultsetLast	Positionnement à la fin d'un ResultSet *Paramètres* : – Identification session – Identification ResultSet. *Retour* : – /
ResultsetNext	Positionnement à l'élément suivant d'un ResultSet *Paramètres* : – Identification session – Identification ResultSet. *Retour* : – /
ResultsetPrev	Positionnement à l'élément précédent d'un ResultSet *Paramètres* : – Identification session – Identification ResultSet. *Retour* : – /
ResultsetGet	Récupération de l'élément courant d'un ResultSet *Paramètres* : – Identification session – Identification ResultSet. *Retour* : – Résultat de l'élément courant du ResultSet.
ResultsetCount	Récupération du nombre d'élément présent dans un ResultSet

	Paramètres : – Identification session – Identification ResultSet. *Retour* : – Nombre d'élément présent à l'instant dans le ResultSet.
ResultsetFree	Destruction d'un ResultSet *Paramètres* : – Identification session – Identification ResultSet. *Retour* : – /

a) Connexion au serveur d'application.

Une connexion avec le serveur d'application doit exister dès que l'utilisateur demande une création de session. Vu le design du protocole, il est tout à fait possible et même souhaitable de partager cette même connexion avec toutes les autres sessions que l'utilisateur pourrait créer. Cela, dans le but de limiter les ressources utilisées tant du côté client que du côté serveur.

Diagramme d'état d'une connexion avec le serveur d'application.



Pour passer de l'état non connecté à l'état connecté et vice versa, les primitives SessionConnect et SessionDisconnect utiliseront le service réseau.

b) Traduction avant envoi des requêtes.

Chaque requête sera décrite dans un container du service sérialisation.
Un ensemble de variables devra décrire le type de requête, ainsi que les paramètres associés à celle-ci.

c) Envoi des requêtes.

L'envoi en lui-même est chose aisée.
Il suffit de transformer chaque container (service sérialisation) en une suite de bytes contigus.

Cette suite de bytes sera alors transmise par l'intermédiaire du service réseau.

d) Récupération des messages du serveur d'application.

Une fois la requête envoyée, on attend une réponse du serveur.
La réponse est reçue sous forme d'une suite de bytes. Cette suite de bytes fera l'objet d'un traitement, décrit au paragraphe suivant.

e) Traduction et présentation des messages reçus.

La suite de bytes reçue étant issue d'un service de sérialisation, il suffit de la lire par l'intermédiaire de celui-ci pour récupérer un container décrivant la réponse.

Ce container sera transmis aux clients pour interprétation.

2. Module Serveur (RSO SRV)

Description :

Ce module est responsable de la bonne marche de l'ensemble du serveur d'application.

On peut découper ce module en trois sous-systèmes :
- Gestion des ressources.
- Gestion des communications.
- Traitement des requêtes

Bien-sûr, ces sous-systèmes utiliseront l'ensemble des services précédemment décrits.

a) Schéma

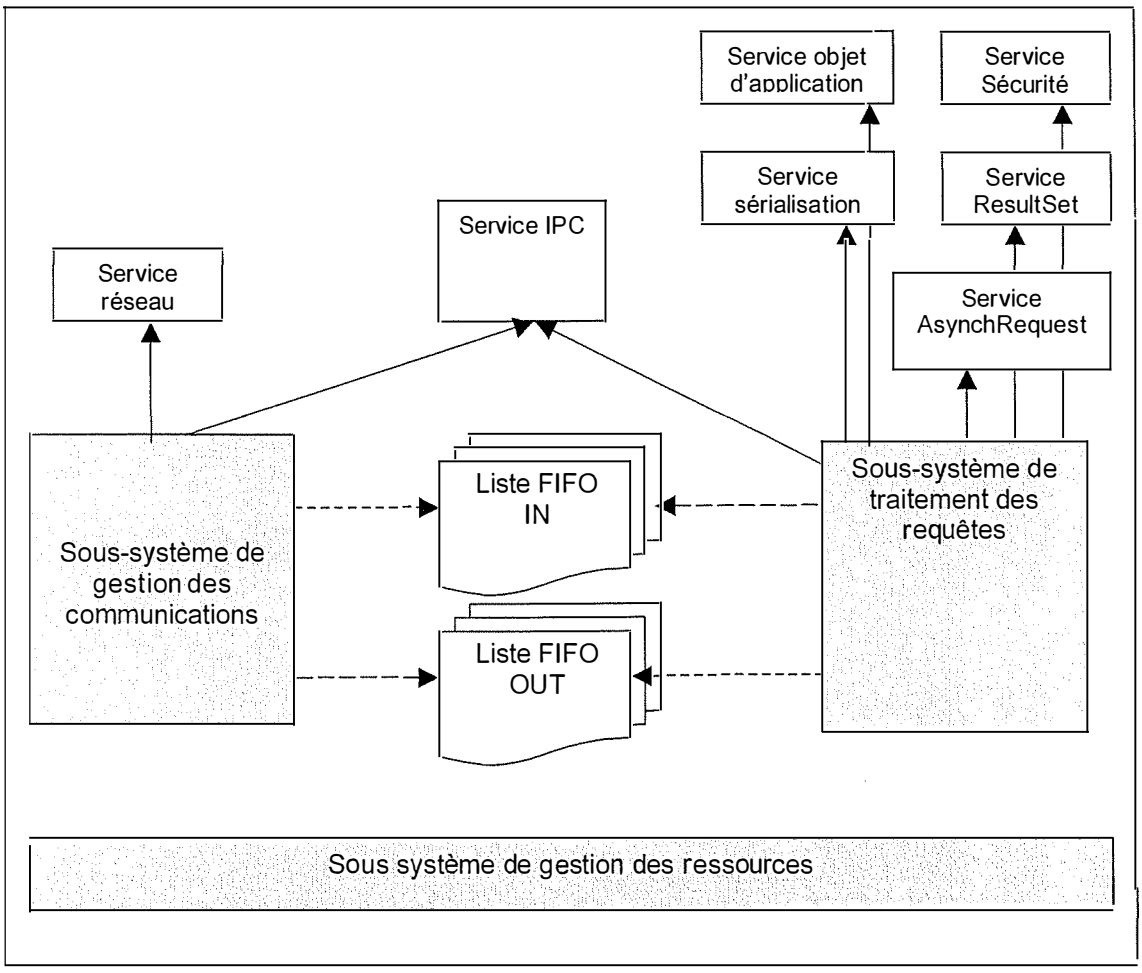


Figure III-7 : Schéma général du serveur

b) Gestion des ressources.

Ce sous-système doit gérer l'ensemble des ressources nécessaires au bon fonctionnement du serveur.

Cela implique :

- Création et libération des ressources systèmes nécessaires. (Structures mémoires)
- Création et libération des services

c) Gestion des communications

Ce sous-système n'intervient en rien dans la logique du système.

-1- Attente de demande de connexions.

Ce sous-système doit mettre en place un processus qui attend des demandes de connexions

-2- Acceptation de connexions

L'acceptation de nouvelles connexions pourra plus tard être sujette à une validation. (Limitation du nombre de connexions, Adresse réseau du client particulière, ...)
Dans un premier temps, toute demande de connexion sera acceptée.
Aucun contrôle n'est fait à ce niveau, ce sous-système gère seulement les connexions physiques.

-3- Transmission des données

Les communications entre la partie réseau du serveur et les autres services se feront en utilisant le service IPC (Liste FIFO).
Il existera donc deux files, une pour les données entrantes et une pour les données sortantes.
Tous les clients connectés au serveur utiliseront donc les mêmes files en entrée et en sortie.

Dès que des données seront envoyées d'un client, ce sous-système placera ces données dans FIFO IN.
A l'inverse, les données à envoyer aux clients seront placées par un autre sous-système dans FIFO OUT. Le sous-système de gestion des communications puisera les données dans cette pile et enverra les données aux clients.

Les données traitées par ce sous-système ne subissent aucune analyse ni traitement.

-4- Terminaison des connexions

Une fois encore, ce sous-système n'intervient pas à un niveau logique. Une déconnexion aura lieu lorsque le client fermera lui-même sa connexion au serveur.

d) Traitement des requêtes.

Par requête, on entend l'ensemble des demandes adressées par un client, pour le serveur par l'intermédiaire du protocole d 'échange définit au début de ce document.

Ce sous-système va obtenir ces requêtes directement du sous-système communication, par l'intermédiaire d'une liste FIFO.

On peut regrouper l'ensemble des primitives du protocole en deux catégories.

- Les primitives de gestion des sessions.
- Les primitives de traitement des OA.

Lorsqu'une requête sera transmise par le sous-système communication, le serveur la décodera. (Le décodage est exactement l'opération inverse du codage réalisé par le service client. (cf. : page 59)

Une fois décodée, la requête sera traitée en fonction de son type.

-1- La gestion des sessions

Pour qu'un client puisse accéder aux différentes ressources du serveur, celui-ci doit s'identifier (SESSION_CONNECT).
Le client fournit un identifiant d'utilisateur, un mot de passe et, éventuellement un identifiant de session.

Après vérification des droits d'accès, deux cas se présentent.

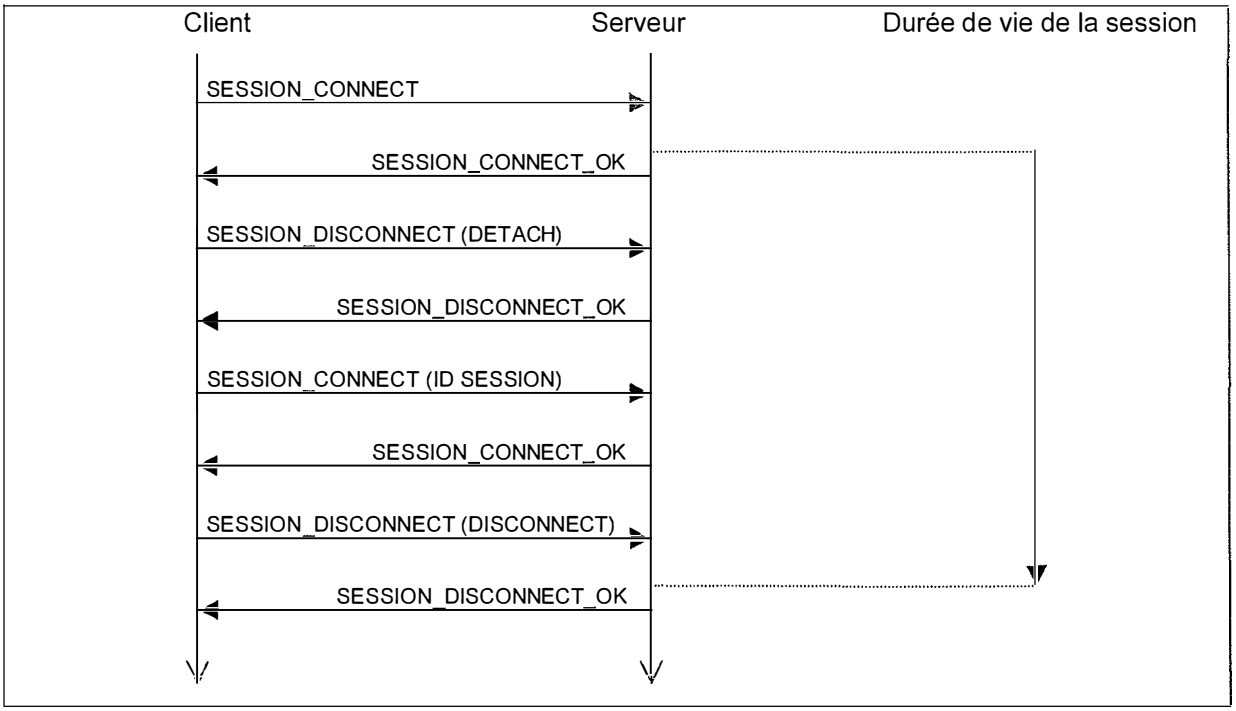
1. Un identifiant de session est fourni et valable, le client est alors attaché à cette session.
2. Un identifiant de session n'est pas fourni, une nouvelle session est alors créée.

Si les droits d'accès ne sont pas valable ou que l'identifiant de session n'existe pas, la demande de connexion est refusée (SESSION_CONNECT_KO) et aucune session n'est créée.

Remarque :

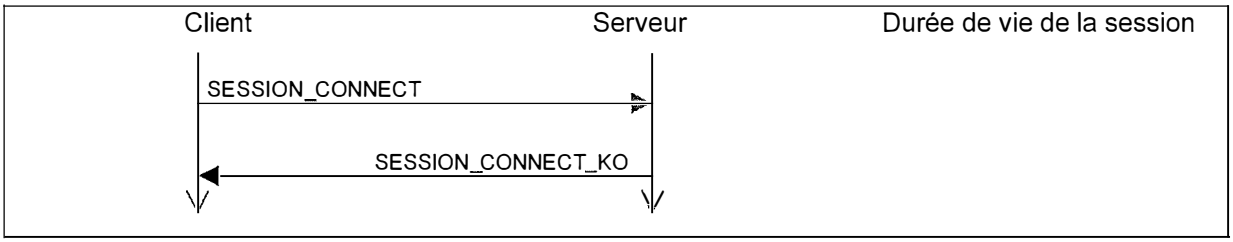
- Une session n'est donc pas limitée par une connexion physique d'un client.
- Une session est limitée à un et un seul utilisateur.

Création d'une session (flux normal)
Ce schéma montre une connexion vers un serveur, un détachement de la session, une reconnexion à une session existante et une déconnexion (Destruction de la session)



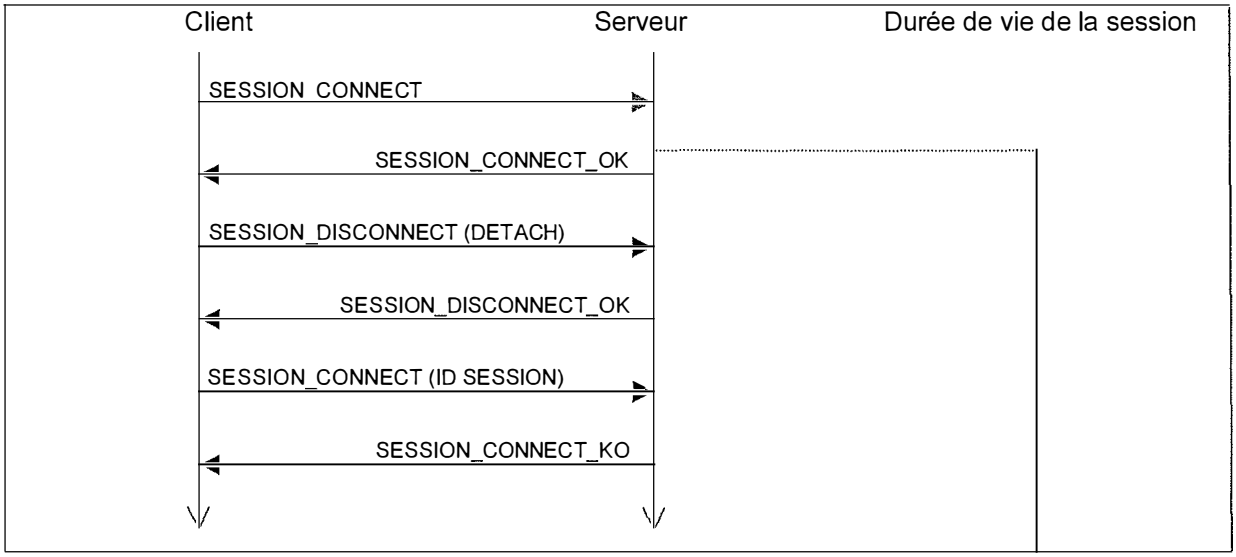
Création d'une session (flux d'exception 1)

Dans ce cas, aucune session n'est créée car la demande de connexion échoue. (Utilisateur inconnu, mot de passe erroné, ...)



Création d'une session (flux d'exception 2)

Ce schéma montre une connexion réussie, suivie d'un détachement. La reconnexion échoue, la session précédemment créée n'est pas détruite. Elle ne le sera que lorsque le propriétaire de celle-ci se reconnectera et demandera explicitement une destruction de la session.



Les échecs de déconnexions ne devraient pas arriver. Toutefois, dans un souci de complétude, ils ont été prévus. Si, un problème survenait lors d'une déconnexion, l'action à prendre dépendra du type d'erreur. Il faut, de toutes façons, essayer de maintenir la session active pour éviter de perdre des données.

-2- *Traitement des OA*

Ce sous-système est très important, ce sera le plus sollicité. C'est pourquoi, il faut y attacher une grande importance en terme de performance.

Toutes les demandes des clients seront traitées par celui-ci; On en distingue trois types :

- Gestion d'instance.
- Exécution de méthode.
- Gestion Resultset

(a) Gestion d'instance.

OA_CREATE	Demande de création d'une instance d'un objet – SOA.CreateInstance *(Nom module, nom'OA, paramètre de création)* – SSM.AddOA *(ID Session, ID OA)* – SNT.Send *(ID Session, OA_CREATE_OK, ID OA)* || SNT.Send *(ID Session, OA_CREATE_KO,…)*

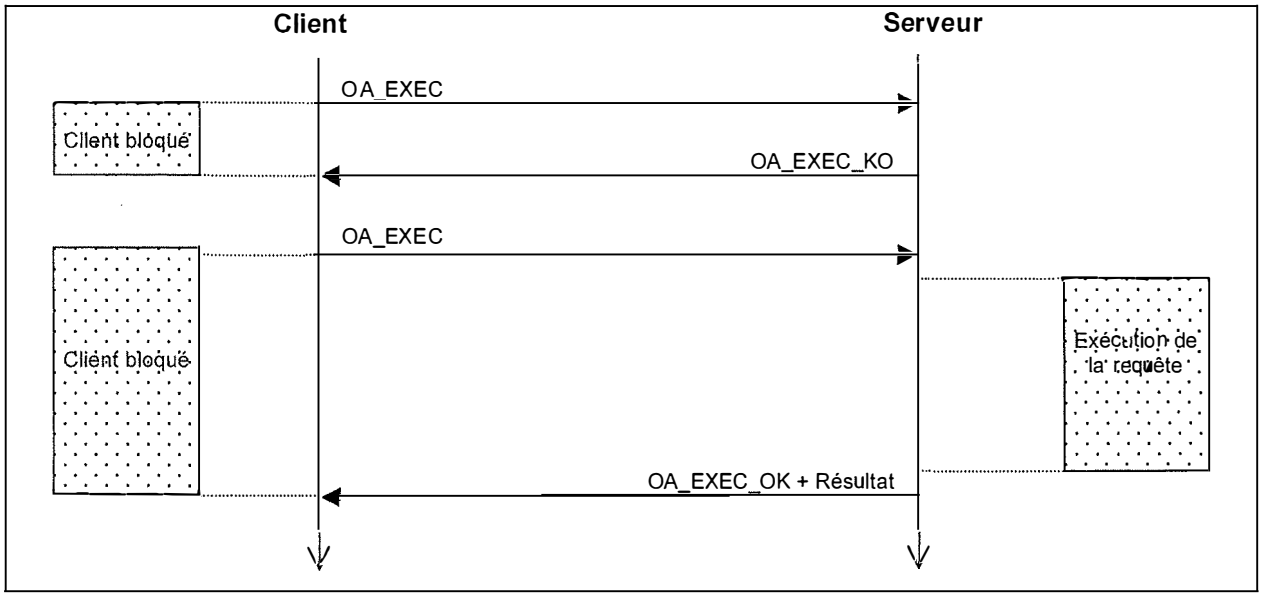
OA_FREE	Demande de destruction d'une instance d'un objet – SSM.DeleteOA *(ID Session, ID OA)* – SOA.FreeInstance *(ID OA)* – SNT.Send *(ID Session, OA_FREE_OK, ID OA)* || SNT.Send *(ID Session, OA_FREE_KO,…)*

(b) Exécution de méthode.

L'exécution d'une méthode peut se faire de deux façons différentes.

- Soit, de façon synchrone.

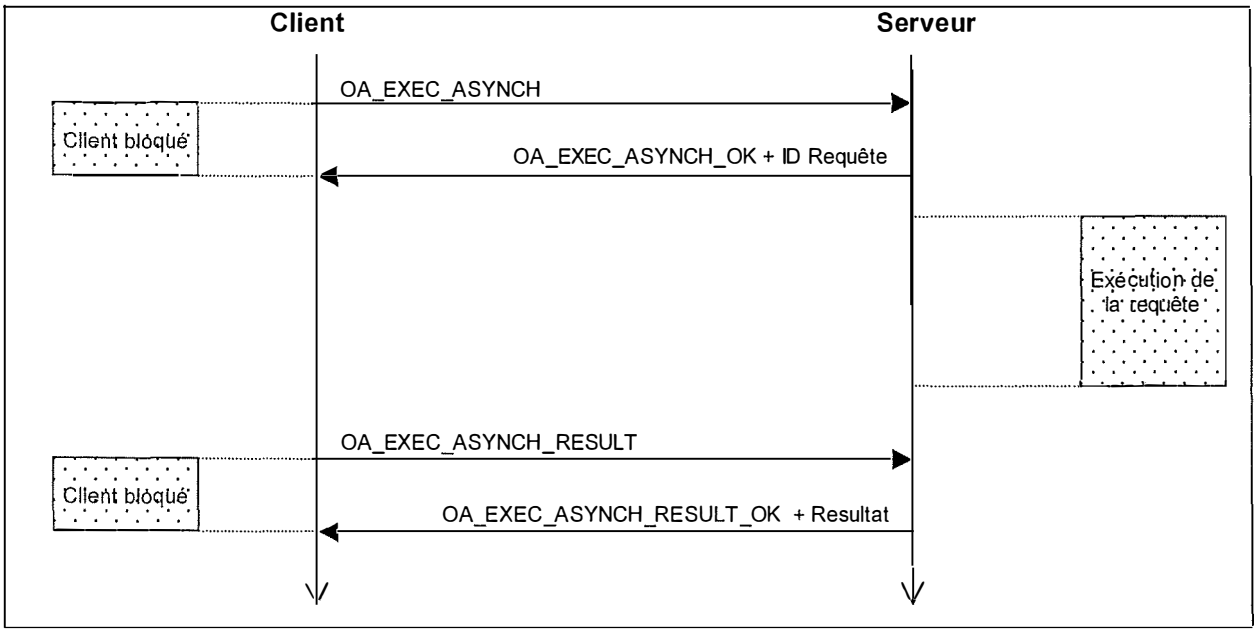
  L'ensemble du processus est séquentiel.
  Le client envoie sa demande, et attend le résultat.



- Soit de façon asynchrone.

  Le client envoie sa demande et n'attend pas la réponse. Toutefois, le serveur doit lui envoyer immédiatement, un numéro de requête qui sera utilisé pour tous les appels subséquents traitant de la même requête.

  Une fois la requête terminée, le client n'est **pas** prévenu. C'est à lui à faire une demande de résultat d'une requête. Si l'exécution n'est pas terminée, un code d'erreur est renvoyé.



Les ressources du serveur étant limitées, viendra un moment où le serveur ne pourra traiter l'ensemble des exécutions en même temps. Le serveur devra donc faire un choix.

Le principe proposé, est de maintenir une liste des méthodes à exécuter, triée sur base de leurs priorités.
Le serveur pourra alors, consulter cette liste pour connaître la requête suivante à exécuter.

Après l'exécution de la méthode, il faut conserver son résultat jusqu'a ce que le client ait récupéré le résultat.

Remarque : La priorité de la requête n'influence que l'ordre d'exécution des méthodes et, en aucun cas, une requête ne sera interrompue pour « céder » la place à une requête plus prioritaire. L'analyse technique tentera de minimiser au maximum ce problème potentiel.

Priorité	Méthode
Haute	M1
Haute	M5
Normale	M4
Basse	M3
Basse	M2

(c) Gestion Resultset

Chaque méthode d'un OA a la possibilité de renvoyer un Resultset. Le concept de ResultSet a été expliqué au chapitre III.A.3 page 25.

Création d'un ResultSet.

Un ResultSet sera créé avant l'exécution d'une méthode d'un OA. A la fin de l'exécution, si le Resultset n'a pas été utilisé, il est détruit. Sinon, son identifiant est envoyé au client.
Dans le cas d'une requête asynchrone, le Resultset n'est pas détruit même s'il est vide puisque l'identifiant est déjà envoyé au client lors de la réponse OA_EXEC_ASYNC_OK.

Le ResultSet Créé est lié à un OA. En aucun cas, une session ne pourra accéder au ResultSet d'un autre OA.

Utilisation d'un ResultSet.

Le client va pouvoir librement parcourir ce ResultSet et récupérer les données dans l'ordre qu'il le désire.

Lors de la demande d'exécution d'une requête, un identifiant est renvoyé directement au client.
Dans le cas d'un appel asynchrone, cela va lui permettre de récupérer des lignes du ResultSet avant même que la méthode ne soit terminée. De cette manière, pour des exécutions lourdes, des résultats partiels pourront être transmis au client.
Les différentes fonctions de parcours ne doivent pas tenir compte de cette fonctionnalité. Si le ResultSet est incomplet et que le client demande d'aller à la dernière ligne de celui-ci, le serveur se positionnera à la dernière ligne du ResultSet incomplet.
C'est au client à vérifier la terminaison de sa méthode par l'intermédiaire du message EXEC_ASYNC_RESULT.

Destruction d'un ResultSet

Un ResultSet sera détruit dans deux cas :
1. Le client se déconnecte définitivement de la session.
2. Le client envoie une demande de destruction du ResultSet

RESULTSET_FIRST	Positionnement sur le premier élément du ResultSet
	– Vérification ID ResultSet appartient à la session – Positionne ResultSet à 1. * Vérification d'usage pour les dépassements de bornes. – Envoi du Résultat requête *(SNT.Send)*

RESULTSET_LAST	Positionnement sur le dernier élément du ResultSet
	– Vérification ID ResultSet appartient à la session – Positionne ResultSet au dernier élément. * Vérification d'usage pour les dépassements de bornes. – Envoi du Résultat requête *(SNT.Send)*

RESULTSET_NEXT	Positionnement sur l'élément suivant du ResultSet
	– Vérification ID ResultSet appartient à la session – Positionne ResultSet à l'élément suivant. * Vérification d'usage pour les dépassements de bornes. – Envoi du Résultat requête *(SNT.Send)*

RESULTSET_PREV	Positionnement sur l'élément précédent du ResultSet
	– Vérification ID ResultSet appartient à la session – Positionne ResultSet à l'élément suivant. * Vérification d'usage pour les dépassements de bornes. – Envoi du Résultat requête *(SNT.Send)*

RESULTSET_GET	Récupération des données pour la position courante du ResultSet
	– Vérification ID ResultSet appartient à la session. – Récupération des données. *(SST.LoadData)* * Vérification d'usage pour les dépassements de bornes. – Envoi du Résultat requête + données *(SNT.Send)*

RESULTSET_COUNT	Récupération du nombre d'élément du ResultSet
	– Vérification ID ResultSet appartient à la session – Récupération du nombre d'élément. *(SST.GetSequenceCount)* – Envoi de ce nombre. *(SNT.Send)* – Envoi du Résultat requête + Nombre *(SNT.Send)*

RESULTSET_FREE	Libération précoce du ResultSet.
	– Vérification ID ResultSet appartient à la session – Effacement du ResultSet. *(SST.RemoveData)* – Envoi du Résultat requête *(SNT.Send)*

G. Les OA d'accès aux bases de données

Il est prévu de fournir un certain nombre de fonctionnalités d'accès aux bases de données. Celles-ci seront implémentées sous forme d'OA. Et cela, au même titre que d'autres OA qui pourraient être écrits par la suite.

Cet accès aux bases de données utilisera la librairie standard de Delphi. Le Borland Database Engine.

Fonctionnalités nécessaires :

- Connexion à une base de données
- Déconnexion d'une base de données.
- Préparation d'un query
- Exécution d'un query.

La gestion du résultat des requêtes SQL n'est pas nécessaire, car, on peut utiliser le concept de ResultSet introduit au niveau du middle-tier.

Chaque instance de l 'OA pourra se connecter à une et une seule base de données à la fois.
Par contre, plusieurs requêtes pourront être exécutées simultanément.

Librairie : AccessDB	
ConnectDB	*Etabli une connexion avec une base de données. Tous les paramètres nécessaires à cette connexion BDE sont passés.*
	Paramètres : – Paramètre de connexion BDE *Résultat* : – Ok | Ko + Code erreur + Message d'erreur
DisconnectDB	*Rompt la connexion établie avec la base de données.*
	Paramètres : – / *Résultat* : – Ok | Ko + Code erreur + Message d'erreur
PrepareQuery	*Prépare le query passé en paramètre au niveau de la base de données et renvoie un identifiant en vue d'une utilisation future.* *Le query peut contenir des paramètres sous la forme communément utilisée :* *« :Nom de paramètre »*
	Paramètres : – Requête SQL *Résultat* : – Ok + Identifiant Query | Ko + Code erreur + Message d'erreur

ExecuteQuery	*Lie les paramètres du query référencé avec ceux passés en paramètres et exécute celui-ci.*
	Paramètres : – Identifiant query – Bind parameters *Résultat* : – Ok + Identifiant ResultSet si SELECT | Ko + Code erreur + Message d'erreur
PrepExecQuery	*Effectue en une seule opération la préparation et l'exécution d'un query*
	Paramètres : – Requête SQL – Bind parameters *Résultat* : – Ok + Identifiant Query + Identifiant ResultSet si SELECT | Ko + Code erreur + Message d'erreur
CloseQuery	*Détruit toutes les structures allouées pour un query.*
	Paramètres : – Identifiant Query *Résultat* : – Ok | Ko + Code erreur + Message d'erreur
Commit	*Effectue un commit sur la transaction courante de la base de données*
	Paramètres : – / *Résultat* : – Ok | Ko + Code erreur + Message d'erreur
Rollback	*Effectue un rollback sur la transaction courante de la base de données*
	Paramètres : – / *Résultat* : – Ok | Ko + Code erreur + Message d'erreur

IV. Analyse technique

A. Environnement technique

Cette analyse sera faite en fonction des données techniques suivantes :

- L'environnement du client et du serveur est Win32.
- Le protocole de communication utilisé est TCP/IP.
- Le langage de développement est le Delphi 5.

Bien que ces contraintes aient été fixées, cela n'empêche pas de prévoir l'emploi de techniques qui faciliteront le portage sur d'autre plate-forme, si le besoin s'en fait sentir.

B. Paradigme de programmation

Delphi étant complètement orienté objet, il serait absurde de ne pas opter pour ce style de programmation.
L'ensemble des services sera donc développé sous forme de classes.

1. Nomenclature

Pour permettre une lecture plus aisée du code, les différents noms de variables, de classes, de fichiers seront normalisés.

1. Une unit comportera une base commune avec la classe qu'elle définit, précédée de 'u'. Le nom de la classe sera, cette même base précédée de 'T' .
   Exemple :

   Unit : uContainer
   Classe : Tcontainer

   Dans le cas où plusieurs classes de même nature seraient définies dans un même fichier, un nom le plus représentatif possible sera utilisé précédé de 'u'

2. Les noms de variable de type simple seront le plus claires possible et seront précédés d'une lettre en minuscule représentant leur type.
   Exemples :

   sName : String
   iCompteur : Integer
   bExist : Boolean
   cchar : Char

3. Les noms de variable de type complexe tel que record, class, … seront précédés de trois lettres minuscules.

   StrListeUtilisateur : TStringList

4. Les constantes seront précédées de 'C' et en majuscules.

   CCONSTANTE : 125

2. Gestion des erreurs

Les erreurs seront gérées exclusivement par des exceptions.
Si une méthode ne renvoie aucun résultat, elle sera déclarée comme procédure. Si une erreur survient, une exception sera lancée.
De même, pour les méthodes renvoyant des résultats, seul des résultats valables sont renvoyés. Les erreurs sont signalées par des exceptions.
L'utilisation des exceptions permet de rendre le code plus clair. Toutefois, les classes de bases de Delphi gérant les exceptions ont une lacune à laquelle il va falloir remédier.
La classe Exception prévoit une propriété de type texte comportant le message d'erreur mais, il n'est pas prévu une propriété de type entier reprenant un code d'erreur. Cela rend l'automatisation de la gestion des erreurs plus ardues.
La classe Exception sera donc héritée par une classe RSOException reprenant un code d'erreur.

Exemple d'héritage de la classe Exception

```
RSOException = Class(Exception)
 Private
  FErrorCode : Integer;
   .
 Public
  Property ErrocCode : Integer Read
FErrorCode Write FErrorCode;
   .
End;
```

De cette classe RSOException sera dérivé un ensemble de classes d'exceptions spécifique à chaque service et module. Cela permettra, premièrement, de spécialiser certaines erreurs et, deuxièmement, de détecter plus facilement d'où provient une erreur lors du débogage.

3. Support thread

Tous les services doivent être développés de manière « Thread-Safe ».
Cela, pour garantir une exécution parfaite des services dans n'importe quel environnement (mono-thread, multi-thread).
Pour rappel, un thread est une unité d 'exécution. Un processus est constitué d'un ensemble de thread partageant le même espace d'adressage.

Des objets de synchronisation fournis par Windows seront utilisés pour y parvenir.
L'objectif de ces objets est d'empêcher que deux threads accèdent en même temps, à la même ressource.

Les objets de synchronisation disponibles sous win32[1] sont les suivants :

- Mutex
- CriticalSection
- Sémaphore
- Event

L'explication de ces divers objets serait trop longue et sortirait du cadre de ce travail. Le lecteur se référera à la documentation de Windows pour comprendre la sémantique de ceux-ci.

Les threads et la librairie Delphi.

Lors de l'utilisation d'objet définit dans la librairie d'objet de Delphi , une attention toute particulière devra être portée pour rendre le code Thread-Safe.
Effectivement, **la librairie d'objet de Delphi n'est pas Thread-Safe**.

[1] Liste non exhaustive.

Les appels des méthodes d'un objet de la VCL devront être protégés, soit par des « CriticalSection » soit par un autre objet de synchronisation si celui-ci est plus adéquat.

Une partie de ces objets sont définis sous forme de classes dans la unit SyncObjs.Pas. L'utilisation de ces classes est conseillée.
Pour les autres objets de synchronisation, il faut utiliser les appels à l'API de Windows définis dans la unit Windows.Pas.

Exemple de protection d'un appel à un objet de la VCL

```
Class aList
.
CS : TCriticalSection ;
.
End ;

Procedure aList.AddElem
Begin
.
 CS.Acquire ;
 List.Add('Ajout dans une TStringList') ;
 CS.Release ;
.
End ;

Function aList.RemoveElem(ID : Integer) ;
Begin
.
 CS.Acquire;
 List.Delete(ID);
 CS.Release;
.
End;
```

4. Débogage

Pour faciliter la découverte de bogue, une unit « uDebug » sera développée.
Cette unit aura pour tâche de gérer une console DOS et un fichier de log.
Une fonction « TRACE » permettant d'afficher des messages et le contenu de variables sera implémentée. Cette fonction effectuera sa sortie dans la console DOS et dans le fichier de Log.

Le fichier log aura pour nom, le nom de l'exécutable en cours de débogage + '.Log'

Cette unit ne devra être compilée que si la variable DEBUG est définie au niveau du compilateur.
Si celle-ci n'est pas positionnée, TRACE référencera une fonction qui ne fait rien. Lors de la compilation, cet appel sera purement et simplement ignoré grâce à l'optimiseur.

C. Le protocole d'échange.

Chaque message envoyé comporte une ou plusieurs parties. Une partie fixe, décrivant la nature du message et ses paramètres et une partie variable comprenant des données que l'on désire transmettre comme les paramètres d'exécution d'une méthode, le résultat d'une exécution, ...

Un message sera donc composé de un ou de deux containers.(voir trois, si nécessaire)

Une solution aurait été de générer un seul container contenant d'autres containers. Pour des raisons de performance (recopie de zone mémoire, ...), cette solution n'est pas retenue.
Ainsi, le serveur ne doit décoder que le container qui l'intéresse.

Message RSO		
Container fixe	Container variable	Container variable

1. Les requêtes

Les requêtes sont toujours composées d'au moins deux variables :

CMD : Integer ->	Définit le type du message.
SESSION : Integer ->	Définit la session à laquelle elles se rapportent.

A ces deux variables vient s'ajouter un ensemble de variables propres à chaque requête

SessionConnect		
Variables	Type	Valeur
CMD	Integer	SESSION_CONNECT = 1
SESSIONID	Integer	
UID	String	
PWD	String	

SessionDisconnect		
Variables	Type	Valeur
CMD	Integer	SESSION_DISCONNECT = 2
SESSIONID	Integer	
MODE	String	Disconnect = 1 Disconnect Force = 2 Detach = 3

OACreate		
Variables	Type	Valeur
CMD	Integer	OA_CREATE = 3
SESSIONID	Integer	
LIB	String	
OA	String	
CONTAINER 2 Paramètres à passer au constructeur de l'objet.		

OAFree		
Variables	Type	Valeur
CMD	Integer	OA_FREE = 4
SESSIONID	Integer	
OAID	Integer	

OAExec

Variable	Type	Valeur
CMD	Integer	OA_EXEC = 5
SESSIONID	Integer	
OAID	Integer	
METHOD	String	
CONTAINER 2, paramètres à passer à la métode		

OAExecAsynch

Variable	Type	Valeur
CMD	Integer	OA_EXEC = 6
SESSIONID	Integer	
OAID	Integer	
METHOD	String	
PRIORITY	Integer	Low = 1 Normal = 2 High = 3
CONTAINER 2, paramètres à passer à la métode		

OAExecAsynchResult

Variable	Type	Valeur
CMD	Integer	OA_EXEC_ASYNCH_RESULT = 7
SESSIONID	Integer	
REQID	Integer	

ResultsetFirst,
ResultsetLast,
ResultsetNext,
ResultsetPrev

Variable	Type	Valeur
CMD	Integer	RESULTSET_FIRST = 8 RESULTSET_LAST = 9 RESULTSET_NEXT = 10 RESULTSET_PREV = 11
SESSIONID	Integer	
RESULTSETID	Integer	

ResultSetGet

Variable	Type	Valeur
CMD	Integer	RESULTSET_GET = 12
SESSIONID	Integer	
RESULTSETID	Integer	

ResultSetCount

Variable	Type	Valeur
CMD	Integer	RESULTSET_COUNT = 13
SESSIONID	Integer	
RESULTSETID	Integer	

ResultSetFree

Variable	Type	Valeur
CMD	Integer	RESULTSET_FREE = 14
SESSIONID	Integer	
RESULTSETID	Integer	

2. Les réponses (MESSAGE_OK)

Ce type de réponse est toujours composé d'au moins deux variables :

CMD : Integer ->	Définit le type du message.
SESSIONID : Integer ->	Définit la session à laquelle elles se rapportent.

A ces deux variables vient s'ajouter un ensemble de variables propres à chaque réponse

SessionConnect		
Variable	Type	Valeur
CMD	Integer	SESSION_CONNECT_OK = 10001
SESSIONID	Integer	

SessionDisconnect		
Variable	Type	Valeur
CMD	Integer	SESSION_DISCONNECT_OK = 10002
SESSIONID	Integer	

OACreate		
Variable	Type	Valeur
CMD	Integer	OA_CREATE_OK = 10003
SESSIONID	Integer	
OAID	Integer	

OAFree		
Variable	Type	Valeur
CMD	Integer	OA_FREE_OK = 10004
SESSIONID	Integer	

OAExec		
Variable	Type	Valeur
CMD	Integer	OA_EXEC_OK = 10005
SESSIONID	Integer	
RESULTSETID	Integer	
CONTAINER 2, Résultat de l'exécution		

OAExecAsynch		
Variable	Type	Valeur
CMD	Integer	OA_EXEC_ASYNCH_OK = 10006
SESSIONID	Integer	
RESULTSETID	Integer	
REQID	Integer	

OAExecAsynchResult		
Variable	Type	Valeur
CMD	Integer	OA_EXEC_ASYNCH_RESULT_OK = 10007
SESSIONID	Integer	
CONTAINER 2, Résultat de l'exécution		

ResultSetFirst, ResultSetLast, ResultSetNext, ResultSetPrev		
Variable	Type	Valeur
CMD	Integer	RESULTSET_FIRST_OK = 10008 RESULTSET_LAST_OK = 10009 RESULTSET_NEXT_OK = 10010 RESULTSET_PREV_OK = 10011
SESSIONID	Integer	
EOF	Integer	False = 0 True = 1
CONTAINER 2, Valeur de la ligne courante du ResultSet		

ResultSetGet		
Variable	Type	Valeur
CMD	Integer	RESULTSET_GET_OK = 10012
SESSIONID	Integer	
EOF	Integer	False = 0 True = 1
CONTAINER 2, Valeur de la ligne courante du ResultSet		

ResultSetCount		
Variable	Type	Valeur
CMD	Integer	RESULTSET_COUNT_OK = 10013
SESSIONID	Integer	
COUNT	Integer	

ResultSetFree		
Variable	Type	Valeur
CMD	Integer	RESULTSET_FREE_OK = 10014
SESSIONID	Integer	

3. Les réponses (MESSAGE_KO)

Le canvas, pour ce type de réponse est le même pour tous les messages, il est composé de quatre variables :

CMD : Integer -> Définit le type du message.
SESSIONID : Integer -> Définit la session à laquelle elles se rapportent.
ErrorCode : Integer -> Code d'erreur
ErrorDetail : String -> Message d'erreur

Erreurs possibles	
Valeur	Description
ERR_INTERNAL = 1	Erreur interne
ERR_SESSION_INVALID = 2	ID Session invalide
ERR_USER_PASS_INVALID = 3	Utilisateur ou mot de passe invalide
ERR_DISCONNECT_INVALID = 4	Mode de déconnexion invalide
ERR_LIBRARY_INVALID = 5	Nom de librairie invalide
ERR_OBJECT_INVALID = 6	Nom d'objet invalide
ERR_OAID_INVALID = 7	ID Objet d'application invalide
ERR_METHOD_INVALID = 8	Méthode inexistante
ERR_REQID_INVALID = 9	ID Requête asynchrone invalide
ERR_RESULTSET_INVALID = 10	ID ResultSet invalide

Liste des réponses KO :

CMD	Codes d'erreurs valides
SESSION_CONNECT_KO = 20001	ERR_INTERNAL ERR_SESSION_INVALID ERR_USER_PASS_INVALID
SESSION_DISCONNECT_KO = 20002	ERR_INTERNAL ERR_SESSION_INVALID ERR_DISCONNECT_INVALID
OA_CREATE_KO = 20003	ERR_INTERNAL ERR_SESSION_INVALID ERR_LIBRARY_INVALID ERR_OBJECT_INVALID
OA_FREE_KO = 20004	ERR_INTERNAL ERR_SESSION_INVALID ERR_OAID_INVALID = 7
OA_EXEC_KO = 20005	ERR_INTERNAL ERR_SESSION_INVALID ERR_OAID_INVALID ERR_METHOD_INVALID
OA_EXEC_ASYNCH_KO = 20006	ERR_INTERNAL ERR_SESSION_INVALID ERR_OAID_INVALID
OA_EXEC_ASYNCH_RESULT_KO = 20007	ERR_INTERNAL ERR_SESSION_INVALID ERR_OAID_INVALID ERR_METHOD_INVALID ERR_REQID_INVALID
RESULTSET_FIRST_KO = 20008	ERR_INTERNAL ERR_SESSION_INVALID ERR_RESULTSET_INVALID
RESULTSET_LAST_KO = 20009	ERR_INTERNAL ERR_SESSION_INVALID ERR_RESULTSET_INVALID
RESULTSET_NEXT_KO = 20010	ERR_INTERNAL ERR_SESSION_INVALID ERR_RESULTSET_INVALID
RESULTSET_PREV_KO = 20011	ERR_INTERNAL ERR_SESSION_INVALID ERR_RESULTSET_INVALID
RESULTSET_GET_KO = 20012	ERR_INTERNAL ERR_SESSION_INVALID ERR_RESULTSET_INVALID
RESULTSET_COUNT_KO = 20013	ERR_INTERNAL ERR_SESSION_INVALID ERR_RESULTSET_INVALID
RESULTSET_FREE_KO = 20014	ERR_INTERNAL ERR_SESSION_INVALID ERR_RESULTSET_INVALID

4. Exemples.

Voici quelques exemples de construction de messages.

Requête avec 1 container

Demande de connexion.

```
SSR.CreateContainer
SSR.SetVariable('CMD',SESSION_CONNECT(1))
SSR.SetVariable('SESSIONID', -1);
SSR.SaveToStream(RSOMESSAGE)
```

Requête avec 2 containers

Exécution d'une méthode.

```
SSR.CreateContainer
SSR.SetVariable('CMD',OA_EXEC(5))
SSR.SetVariable('SESSIONID',SessID)
SSR.SetVariable('OAID',OAID)
SSR.SetVariable('METHOD','UpdateClient')
SSR.SaveToStream(RSOMessage)

SSR.CreateContainer
SSR.SetVariable('Nom','Coyette')
SSR.SetVariable('Prenom','Laurent')
SSR.SaveToStream(RSOMessage)

Le message envoyé au serveur sera :
```

Cntr 1 Ses, CMD, OAID, METHOD	Cntr 2 NOM, PRENOM

Réponse (OK)

Réponse à une demande de connexion

```
SSR.CreateContainer
SSR.SetVariable('CMD',OA_CONNECT_OK(10001))
SSR.SetVariable('SESSIONID',SessID)
SSR.SaveToStream(RSOMessage)
```

Réponse (KO)

Réponse à une demande de connexion

```
SSR.CreateContainer
SSR.SetVariable('CMD',OA_CONNECT_KO(20001))
SSR.SetVariable('SESSIONID',SessID)
SSR.SetVariable('ERRORCODE',ERR_USER_PASS_INVALID)
SSR.SetVariable('ERRORDETAIL','Utilisateur Coyette invalide')
SSR.SaveToStream(RSOMessage)
```

D. Analyse des services

1. Service structuration de données (SSD)

a) Outil Liste

Cette liste s'appuie presque entièrement sur l'objet TStringList de la VCL. Cela est voulu pour faciliter le développement et éviter la réécriture d'algorithmes existants.

La plus-value de cette classe est double :

1. Faciliter les maintenances futures en donnant la possibilité de rajouter des fonctionnalités à cette liste sans devoir re-parcourir l'ensemble du code pour remplacer les objets TStringlist par une nouvelle classe.
2. Rendre cette liste thread-safe pour permettre une utilisation aisée des listes dans l'environnement du serveur.

Pour rendre cette liste thread-safe, un objet de synchronisation windows de type CriticalSection sera utilisé.
Chaque appel susceptible de modifier ou de lire des valeurs sera encapsulé dans une critical section.
L'utilisation d'une section critique est détaillée au chapitre IV.B.3 page 71.

TStringListTS
Méthodes

Create
Initialisation TstringList

Free
Destruction TstringList

Add(Text : String) : Integer
Ajout d'un élément à la liste. La position du nouvel élément est renvoyée.

AddObject(Text : String ; Data : Pointer) : Integer
Ajout d'un élément à la liste. Un pointeur est associé à cet élément. La position du nouvel élément est renvoyée.

Insert(Pos : Integer ; Text : String) : Integer
Insertion d'un nouvel élément à la position indiquée par Pos dans la liste.
La position des éléments supérieurs à Pos est incrémentée de 1.
La position du nouvel élément est renvoyée.

InsertObject(Pos : Integer ; Text : String ; Data : Pointer) : Integer
Insertion d'un nouvel élément à la position indiquée par Pos dans la liste. Un pointeur est associé à cet élément.
La position des éléments supérieurs à Pos est incrémentée de 1.
La position du nouvel élément est renvoyée.

Delete(Index : Integer)
Suppression de l'élément ayant comme position, index
La position de tous les éléments peut être altérée.

Strings[Index : Integer] : String
Retourne le texte associé à l'élément se trouvant à la position Index dans la liste.

Objects[Index : Integer] : Pointer
Retourne le pointeur associé à l'élément se trouvant à la position Index dans la liste.

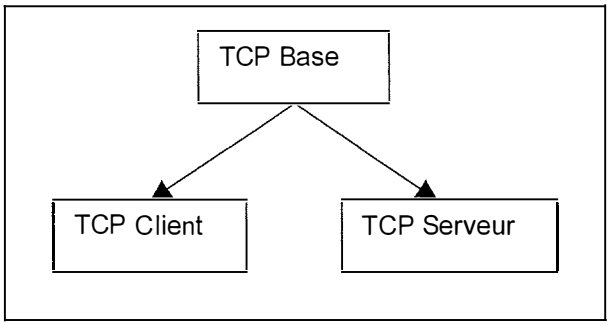
IndexOf(Texte : String) : Integer
Renvoie la position du premier élément ayant Texte comme texte associé.

Count : Integer
Renvoie le nombre d'éléments de la liste.

2. Service Réseau (SNT)

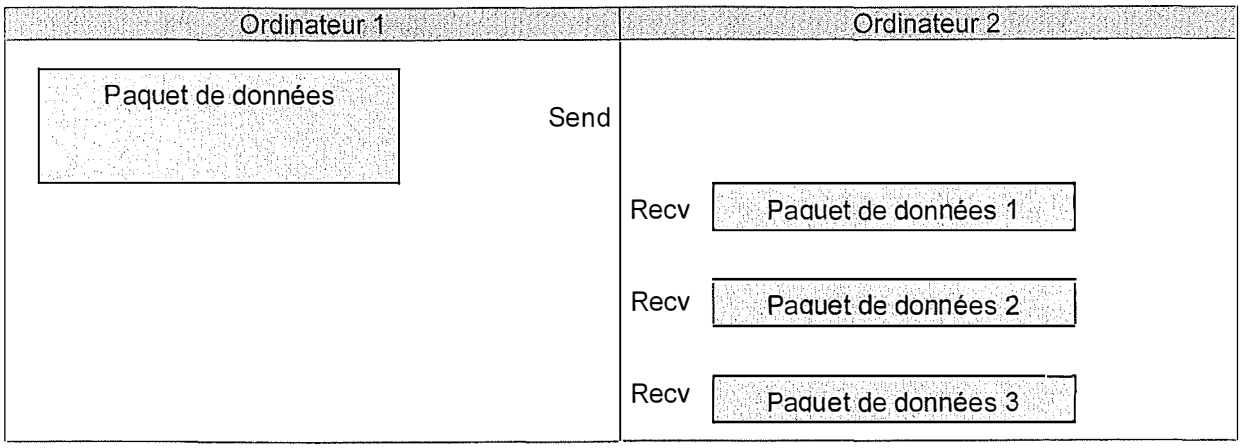
L'ensemble de ce service se repose sur la couche de communication Winsock 2 de Windows. Certaines fonctions de l'API winsock 2 ne sont pas définies au niveau de Delphi 5. Il suffit, de les déclarer ou, d'utiliser des déclarations que l'on trouve très facilement sur internet. Le meilleur portage de l'API que l'on peut, à mon sens, trouver est celui du projet « JEDI » (http://delphi-jedi.org)

Pour résumer la situation, nous avons besoin d'un support de communication performant, fiable et capable de s'adapter à un environnement soit client, soit serveur.
Le design d'un seul et même composant pour approcher ces deux aspects ne me semble pas être une bonne solution.
Le composant fonctionnant dans l'environnement serveur devra utiliser certaines techniques pour permettre la gestion d'un grand nombre de connexions. Ces techniques, si elles sont implémentées au niveau du client, alourdiront celui-ci sans raison.
C'est pourquoi, il me semble judicieux d'aborder ce service en scindant les deux environnements. Toutefois, les services de base restent les mêmes. Nous allons donc découper ce service en trois classes. Une classe de base commune, une classe client et une classe serveur.



a) Format des messages.

Un point important à souligner, découvert lors de l'analyse du fonctionnement du protocol TCP/IP et de son implémentation sous Win32, est que ce protocol n'est pas « **Message Based »** mais bien « **Stream Based »**. (Du moins, sous Win32)
Un paquet de données envoyé (par un appel API) au travers un réseau par tcp ne sera pas forcément reçu sous forme d'un seul paquet de données. Il sera **peut-être** réparti sur plusieurs paquets.
A l'inverse, l'envoi de plusieurs paquets donnera peut-être lieu à une seule réception.
Cela va dépendre de beaucoup de choses telles que la taille des data, la tailles des buffers internes,...



Etant donné le protocole d'échange défini, nous devons implémenter des fonctions permettant de se sortir de ce mauvais pas. L'envoi d'un message, doit correspondre à la réception de ce même message. Ce service sera donc chargé de rassembler les données en unités logiques compréhensibles par les couches supérieures du programme.

Pour pouvoir synchroniser les messages reçus, nous devons connaître la taille du message envoyé. Or, le système ne peut nous le dire puisqu'il l'a peut-être coupé ou qu'il en a assemblé plusieurs. Pour permettre cela, la taille du message sera placée dans le message. Le service pourra alors lire cette taille et construire son message à partir de celle-ci.

Header = 'RSO'	Taille du message (Unsigned 32 bit)	Données

- Header : permettra de garantir que l'on se trouve bien au début d'un message et éventuellement, permettra de se re-synchroniser
- Taille du message : elle sera codée sur 32 bits non-signés. Cela permet donc théoriquement l'envoi de message de 4 Gb. Ce qui est, pour l'instant tout à fait illusoire.
- Données : l'ensemble des données sans modification aucune.

b) Stockage des données reçues

L'échange des données entre ce service et les autres services doit être le plus souple possible. Le problème est que l'on ne connaît pas à l'avance la taille des messages que l'on va recevoir.

Le service fournira la possibilité de stocker le message, soit :

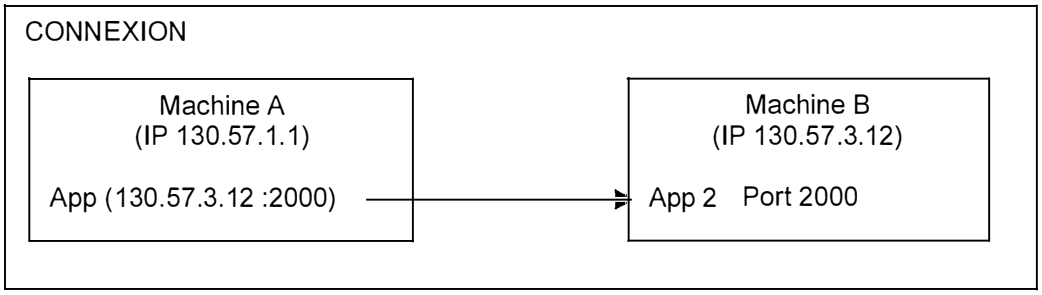
- Dans une zone mémoire préalablement allouée. Si le message est trop grand pour la zone allouée, celui-ci sera tout simplement perdu et un message d'erreur sera retourné.
- Dans un objet TStream de la VCL de Delphi. Cet objet offre l'avantage de pouvoir stocker des suites de bytes de longueur variable. De plus, il existe des spécialisations de ces classes permettant de choisir le média de stockage. (Mémoire: TMemoryStream, Fichier: TFileStream,...)

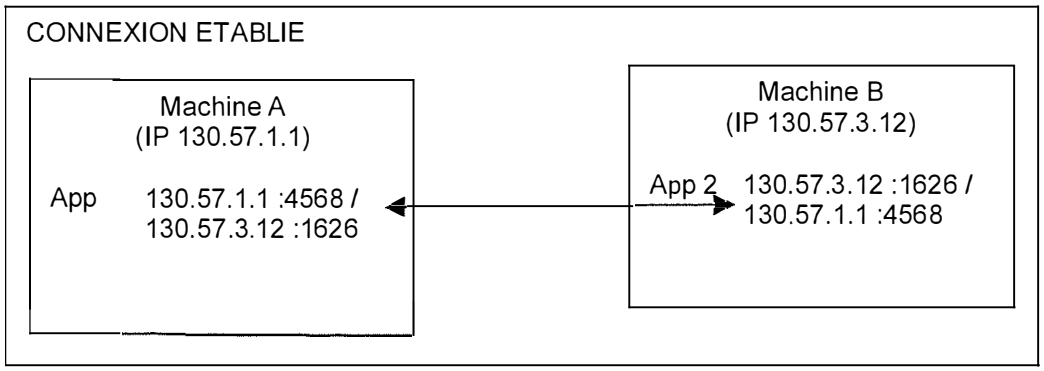
c) Principe de base d'une connexion TCP/IP

Il existe deux méthodes pour faire communiquer deux applications en TCP/IP. L'une est dite sans connexion (Datagramme) et l'autre avec connexion (Socket stream). Expliquer plus en détail ce point sortirait du cadre de ce travail.

Nous utiliserons la deuxième méthode car, celle-ci garantit l'arrivée de tous les paquets. Les datagrammes eux, ne sont pas fiables. Ils sont plutôt utilisés dans des applications multimédias où la perte d'un paquet n'est pas grave.

Une machine A se connecte avec une application fonctionnant sur une machine B en utilisant l'adresse IP de la machine B ainsi qu'un numéro de port. Ce numéro identifie l'application qui doit recevoir la connexion.
Une fois la connexion effectuée, celle-ci est identifiée par une adresse IP/Port sur chaque machine. (Remarque le port attribué pour la connexion est différent de celui utilisé pour l'établir). Cette association est appelée **Socket**.





Un point de connexion sera donc identifié par un Socket. L'ensemble de l'API Winsock fonctionne avec ce Socket.

Une connexion est donc identifiée sur une machine par deux sockets. Le socket du point de connexion local et le socket du point de connexion distant.

d) TCP base

Classe TTCPBase

Cette classe va fournir les services de base pour les deux autres classes. Nous allons encapsuler l'ensemble des API de winsock et simplifier ainsi les tâches de base telles que la connexion, déconnexion, envoi et réception de données.

La création, l'envoi et la réception des messages tels que décrits au paragraphe IV.D.2.a) Page 80 seront pris en charge par les classes TTCPClient et TTCPServer. Ces tâches seront approchées dans les deux classes, d'une manière radicalement différente ; cela dans un souci d'optimalisation.

De même, la gestion de l'arrivée des évènements sur les sockets sera totalement prise en charge par les deux classes filles.

TTCPBase
Propriétés
WsaData : TWsaData
Structure nécessaire pour l'initialisation de la couche Winsock. Le lecteur se référera à la documentation Winsock pour le contenu de celle-ci.
Méthodes
Create
Initialisation de la couche Winsock
Connect (Var Socket : TSocket ; Host : String; Port :Integer);
Connexion à une machine identifiée par Host(Adresse IP ou Nom de machine):Port
Si celle-ci est établie, son identifiant est retourné dans Socket.
Disconnect (Socket : TSocket);
Fermeture de la connexion identifiée par Socket.
Listen (Var Socket : TSocket; Port : Integer);
Prévient la couche Winsock que l'on attend maintenant des connexions.
- Socket : Identification du point local.
- Port : Port sur lequel on attend des connexions.

Accept (ServerSocket : TSocket ; Var ClientSocket : TSocket)
Accepte une demande de connexion.
- ServerSocket : Socket sur lequel est arrivé la demande.
- ClientSocket : Point de connexion distant établi.

Send(Socket : TSocket ; Buffer : Pointer ; Size : DWord; Flags : Integer)
Envoie un paquet de données.
- Socket : Point distant pour l'envoi du message.
- Buffer : Pointeur sur les données à envoyer.
- Size : Taille des données en bytes.
- Flags : Paramètre éventuel d'envoi. (Le lecteur se référera à la documentation Winsock)

SendMessage(Socket : TSocket ; Buffer : Pointer ; Size : DWord; Flags : Integer)
Envoie un message tel que décrit au paragraphe IV.D.2.a) Page 80.
- Socket : Point distant pour l'envoi du message.
- Buffer : Pointeur sur les données à envoyer.
- Size : Taille des données en bytes.
- Flags : Paramètre éventuel d'envoi. (Le lecteur se référera à la documentation Winsock)

Recv(Socket : TSocket ; Buffer : Pointer ; MaxLen : DWord; Flags : Integer) :DWord
Reçoit un paquet de données.
- Socket : Point distant pour lequel on veut lire un message.
- Buffer : Zone mémoire fournie pour stocker les données reçues. (Attention, cette zone doit avoir été allouée au préalable)
- MaxLen : Taille maximum des données que l'on peut recevoir. Si celle-ci est plus grande, une exception est générée.
- Flags : Paramètre éventuel de réception. (Le lecteur se référera à la documentation Winsock)

Retour : La longueur des données reçues.

RecvMessage(Socket : TSocket ; Buffer : Pointer ; MaxLen : DWord; Flags : Integer) :DWord

Reçoit un message tel que décrit au paragraphe IV.D.2.a) Page 80.

- Socket : Point distant pour lequel on veut lire un message.
- Buffer : Zone mémoire fournie pour stocker les données reçues. (Attention, cette zone doit avoir été allouée au préalable)
- MaxLen : Taille maximum des données que l'on peut recevoir. Si celle-ci est plus grande, une exception est générée.
- Flags : Paramètre éventuel de réception. (Le lecteur se référera à la documentation Winsock)

Retour : La longueur des données reçues.

SendMessageStream(Socket : TSocket ; Stream : TStream ; Flags : Integer)

Envoie un message tel que décrit au paragraphe IV.D.2.a) Page 80.

- Socket : Point distant pour l'envoi du message.
- Stream : Objet stream contenant les données à envoyer.
- Flags : Paramètre éventuel d'envoi. (Le lecteur se référera à la documentation Winsock)

RecvMessageStream(Socket : TSocket ; Stream : TStream ; Flags : Integer) :DWord

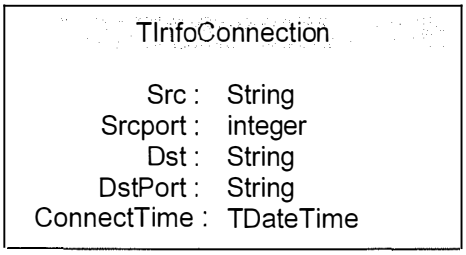
Reçoit un message tel que décrit au paragraphe IV.D.2.a) Page 80.

- Socket : Point distant pour lequel on veut lire un message.
- Stream : Objet stream prêt à recevoir les données du message.
- Flags : Paramètre éventuel de réception. (Le lecteur se référera à la documentation Winsock)

Retour : La longueur des données reçues.

GetConnectionInfo(Socket : TSocket ; Var ConnectionInfo : TConnectionInfo)

Récupère les informations de la connexion identifiée par Socket et remplit une structure TInfoConnection passée en paramètre.



e) TCP Client

-1- Gestion des évènements.

Il existe plusieurs méthodes pour détecter l'arrivée d'évènements sur une connexion. Dans le cas de la classe client, nous allons utiliser la fonction Select de l'API. Une autre méthode sera utilisée pour la classe Serveur.

La fonction Select permet d'obtenir les évènements arrivés sur un ou plusieurs Sockets. Cette fonction permet également de spécifier un Timeout.

-2- Envoi et réception des messages.

Dans le cas du client, le problème est relativement simple. L'environnement étant mono-tâche, les demandes peuvent êtres traitées séquentiellement.

- Envoi de message : Tant que toutes les données n'ont pas été envoyées, on boucle. On rend la main une fois le message totalement envoyé.
- Réception de message : Tant que l'ensemble des données d'un message n'a pas été reçu, on répète le processus. Dès qu'un message entier a été reçu, on rend la main à la routine appelante. Et ce, même s'il existe encore des données disponibles. Cette lecture sera prise en charge lors d'une autre demande de réception.

-3- Classe TTCPClient

Cette classe ne comportera pas beaucoup de valeur ajoutée. En effet, la classe de base implémente la plupart des fonctionnalités nécessaires.

TTCPClient
Propriétés
Connection : TSocket
Identifiant de point de connexion distant. (TTCPServer)
Remarque : L'analyse fonctionnelle définit clairement que le service client doit pouvoir gérer des connexions multiples. Toutefois, ce nombre ne sera jamais élevé. Il est donc préférable de concevoir la classe mono connexion et ainsi, de la garder la plus simple possible. Il suffira de gérer autant d'instance de cette classe que l'on ne voudra de connexion.
Méthodes
Connect (Host : String ; Port : Integer) ;
Etablit une connexion avec la machine identifiée par Host sur le port identifié par Port.
Le socket du point de connexion distant est stocké dans la propriété Connection.
Connection := TTCPBase.Connect(Host, Port)
Disconnect
Fermeture de la connexion identifiée par la propriété Connection.
TTCPBase.Disconnect(Connection)
SendMessage(Buffer : Pointer ; Size : Dword)
Envoie un message tel que décrit au paragraphe IV.D.2.a) Page 80.
- Buffer : Pointeur sur les données à envoyer.
- Size : Taille des données en bytes.

RecvMessage(Buffer : Pointer ; MaxLen : DWord) :DWord

Reçoit un message tel que décrit au paragraphe IV.D.2.a) Page 80.

- Buffer : Zone mémoire fournie pour stocker les données reçues. (Attention, cette zone doit avoir été allouée au préalable)
- MaxLen : Taille maximum des données que l'on peut recevoir. Si celle-ci est plus grande, une exception est générée.

Retour : La longueur des données reçues.

SendMessageStream(Stream : Tstream)

Envoie un message tel que décrit au paragraphe IV.D.2.a) Page 80.

- Stream : Objet stream contenant les données à envoyer.

RecvMessageStream(Stream : Tstream) :DWord

Reçoit un message tel que décrit au paragraphe IV.D.2.a) Page 80.

- Stream : Objet stream prêt à recevoir les données du message.

Retour : La longueur des données reçues.

WaitforEvent(TimeOut : DWord) : Integer

Attend l'arrivée d'un événement sur la connexion identifiée par la propriété Connection.
Après TimeOut (Millisecondes) la fonction se termine, même si aucun événement n'est arrivé.

Retour :

- -1 si le TimeOut a expiré.
- FD_READ : Un message est disponible.
- FD_CLOSE : la connexion a été fermée par le point distant.
- FD_CONNECT : La demande de connexion a été acceptée par le serveur.
- FD_WRITE : Le socket peut envoyer des données.

(FD_XXX sont des constantes définies dans l'API Winsock)

GetConnectionInfo(Socket : TSocket ; Var ConnectionInfo : TConnectionInfo)

Appel la méthode TTCPBase.GetConnectionInfo(Socket, ConnectionInfo)

f) TCP Serveur

-1- Gestion des évènements

Comme dit précédemment, il existe plusieurs méthodes pour détecter la présence d'événements arrivés sur un ou plusieurs sockets.
La méthode utilisée pour la classe TTCPClient (Select) ne peut être utilisée dans ce cas. En effet, cette fonction permet de tester un maximum de 64 Sockets à la fois. Ceci est tout à fait insuffisant pour notre serveur. Cette méthode obligerait à appeler séquentiellement la fonction select par groupe de 64 Sockets.

Heureusement, une autre méthode existe. Un ou plusieurs sockets peuvent être associé à un event (Objet de synchronisation de Win32). Il suffit alors de tester cet event pour savoir si un événement est disponible sur un socket. Une autre fonction permet de connaître le numéro du socket sur lequel est arrivé cet événement (Fonction relativement lourde car il faut l'appeler pour chacun des sockets associés à l'event signalé).
Nous allons donc attendre l'arrivée d'un événement par l'intermédiaire d'une fonction Wait de Win32 (Ressource CPU quasi nulle).
De plus, les fonctions Wait de Win32 permettent d'attendre simultanément sur plusieurs events. Cela va permettre de répartir nos sockets sur plusieurs events pour accélérer la recherche du socket sur lequel est arrivé cet événement ; Le défaut de cette méthode est que lorsque l'event passe à l'état signalé, il faut interroger chaque socket connecté à cet event pour savoir s'il a recu un event TCP ou pas.

Nous trouvant dans un environnement serveur qui sera vraisemblablement multi-thread.
Nous allons maintenir une liste des événements reçus. Cette liste sera implémentée sous forme d'une classe dérivée de TList. Ce qui va permettre d'y ajouter le support multi-thread. Plusieurs threads pourront alors consulter cette liste et ainsi réagir à ces events.

Les événements pouvant être reçus sont les suivants :

FD_ACCEPT	Une demande de connexion est arrivée
FD_CONNECT	Une demande de connexion a été acceptée. Cet événement ne doit pas être géré puisque le serveur n'implémente pas de fonction de connexion.
FD_CLOSE	Une demande de fermeture de la connexion est arrivée
FD_READ	Des données sont disponibles à la lecture.
FD_WRITE	Signale que l'on peut écrire sur le socket en question. Cet événement peut éventuellement être traité pour savoir si l'on peut écrire sur le socket plutôt que d'essayer d'écrire et de se rendre compte que ce n'est pas possible pour l'instant.
FD_OOB	Signale l'arrivée de donnée "Out of Band". Cette fonctionnalité n'est pas gérée par le serveur. Cet événement peut donc être ignoré.

Classe TTCPEventList

Remarque : le code doit être Thread-Safe.

TTCPEventList

Méthodes

AddEvent(Socket : TSocket ; Event : Integer)

Ajoute un événement à la liste

getEvent(Var Socket : TSocket ; Var Event : Integer)

Donne le premièr événement de la liste et le retire de celle-ci.

- Socket : Le socket qui a reçu l'événement.
- Event : Le type d'événement. (FD_READ, FD_CLOSE, FD_ACCEPT,FD_WRITE)

WaitForEvent(Timeout : DWord) : Boolean

Attend l'arrivée d'un événement dans la liste pour une durée maximum de Timeout.

Retour :

- True : Au moins un événement est présent dans la liste.
- False : Le timeout a expiré.

-2- *Envoi et réception des messages.*

Contrairement à un client, nous devons faire face à la réception et à l'envoi de plusieurs messages en même temps. Et ce, avec des ressources limitées. Utiliser la même technique que pour le client risquerait d'engorger le stack TCP du serveur et de bloquer temporairement les clients.
Effectivement, le stack TCP est forcé de nous donner des données que, peut-être, il n'a pas encore reçues. Un temps précieux sera alors perdu à attendre que ces données soient disponibles. Par contre, un autre client aura peut-être envoyé des données qui elles, seront déjà disponibles. De plus, les ressources du stack n'étant pas illimitées, à un certain moment, les buffers de celui-ci seront pleins et plus personne ne pourra envoyer de demande.

La réception de messages

Nous aurions pu traiter chaque message dans son entierté et faire fonctionner N Threads chargés de l'envoi des messages.
Cette solution permettait de ne servir simultanément que N messages. De plus, la synchronisation entre les threads devenait compliquée pour s'assurer que les morceaux de messages reçus ne soient pas mélangés.

La meilleure solution semble être celle où l'on se plie au fonctionnement du système. Lorsque des données sont disponibles, on les lit (à concurrence de la taille d'un buffer). Ensuite, on passe à une autre connexion ayant reçu des données. Lorsque nous avons reçu assez de données pour reconstituer un message dans son entièreté, celui ci pourra être récupéré par la couche supérieure, en vue d'être traité.
Cette solution va permettre de rentabiliser au maximum le système. L'envoi de messages conséquents ne pénalisera pas l'ensemble des clients en train d'envoyer ou de recevoir des messages.

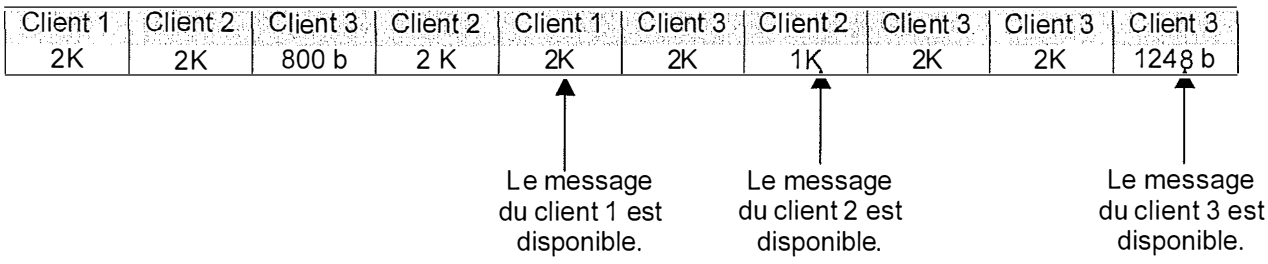
Bien-sûr, cette solution nécessite un plus gros travail de la part de la classe TTCPServer. Elle doit garder, pour chaque client, une liste dynamique des messages en cours de réception et une liste de ceux non encore récupérés pour traitement.

Pour éviter les problèmes de synchronisation évoqués plus haut, un seul thread sera chargé de la réception des messages. Comme ce processus est prévu pour traiter séquentiellement des petits morceaux de message, plusieurs threads seraient inutiles.

Exemple :

Soit, 3 clients envoyants des messages de respectivement 4K, 5K et 8K.
Buffer interne du serveur : 2K.

Le serveur lira séquentiellement les messages selon le schéma suivant :
Attention, ce schéma peut changer car nous réagissons au events de la couche TCP.



Bien que les réceptions soient traitées séquentiellement, chaque message est reçu dans un ordre relativement logique, en fonction de sa taille.

Classe TsocketListMessageIn dérivée de Tlist.

Attention, la réception se faisant par un seul thread ; il ne faut pas se préoccuper de l'ordre de réarrangement des messages. Toutefois, la récupération des messages complets par l'intermédiaire de GetMessage se fera par un autre thread. Il faut donc gérer cela par l'intermédiaire d'une section critique. Cette méthode empêchera l'exécution simultanée des deux fonctions suivantes.

TSocketListMessageIn

Méthodes

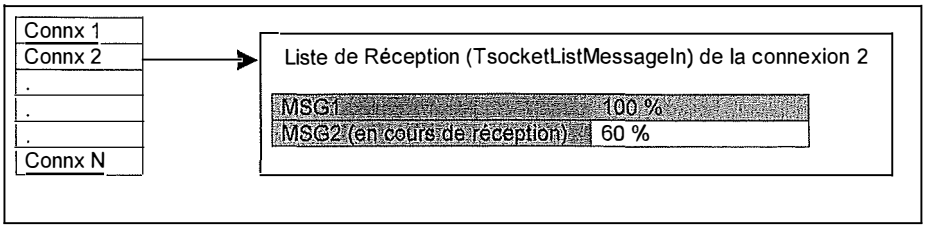
AddChunkMessage(Buffer : Pointer ; BufSize : Dword)

Constitue les messages en concaténant les morceaux reçus dans buffer. La méthode se basera sur le header du message pour composer correctement les messages.

GetMessage : TmemoryStream

Cette méthode renverra un objet TMemoryStream correspondant au premier message reçu complètement. Une fois le message donné, celui-ci est effacé de la liste.
Si aucun message n'est complet, la méthode renvoi Nil.

A chaque connexion active sera associé une telle liste qui conservera les messages en cours de réception et ceux non transférés à la couche supérieure.



L'envoi de messages

L'envoi de messages peut être vu de la même façon. Le serveur enverra séquentiellement des morceaux de messages pour donner à chaque client la même chance de recevoir ses données.

La seule différence est que là, on ne réagit plus à des events de la couche TCP. C'est le serveur qui, en interne doit se « souvenir » que le message n'a pas encore été envoyé complètement et qu'il devra continuer une fois qu'il aura « fait le tour » des connexions nécessitant l'envoi de données.

La gestion de cette liste est moins compliquée que la réception vu qu'il ne faut pas faire la découpe en messages ; il suffit d'implémenter une liste contenant la liste des objets TmemoryStream à envoyer à un client donné.
Comme la classe TMemoryStream conserve la position courante de lecture ou d'écriture, à chaque passe, on pourra repartir de cette position.

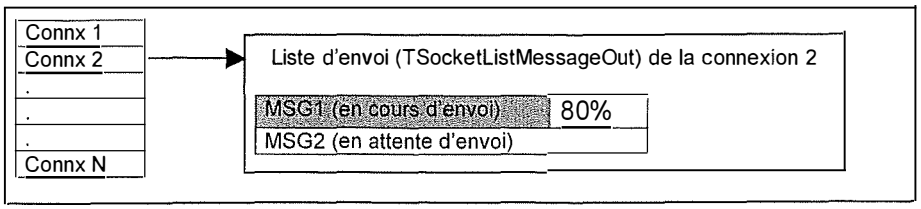
A chaque connexion active sera associé une telle liste qui conservera les messages en cours d'envoi et ceux en attente d'envoi.
Comme pour la réception, un seul thread sera chargé d'effectuer l'envoi pour ne pas devoir supporter une synchronisation lourde et inutile.

Classe TsocketListMessageOut dérivée de Tlist.

Comme pour la classe TsocketListMessageIn, une attention toute particulière doit être portée pour éviter l'exécution simultanée de méthodes par le thread chargé de l'envoi des messages et celui plaçant les messages à envoyer dans la liste.

TSocketListMessageOut
Méthodes
AddMessage(Message : TMemoryStream)
Place dans la liste un nouveau message. (Ce message est placé en fin de liste)
getcurrentMessage : TMemoryStream
Renvoie une référence sur le message en cours d'envoi. (Toujours le premier de la liste)
RemoveCurrentMessage
Supprime le premier message de la liste.

A chaque connexion active sera associé une telle classe.



-3- Identification des connexions

Au vu des besoins nécessaires à l'envoi et à la réception de messages, chaque connexion nécessitera quelques informations supplémentaires. Celles-ci seront placées dans une classe TServerSocketElem.

TserverSocketElem : TSocketElem

Proriétés

Socket : TSocket

Identifiant de la connexion

MessageIn : TSockeListMessageIn

Liste des messages reçus ou en cours de réception sur cette connexion

MessageOut : TSocketListMessageOut

Liste des messages à envoyer ou, en cours d'envoi vers cette connexion

-4- Classe *TTCPServer*

Il est nécessaire de conserver un ensemble d'attributs propres à chaque connexion. C'est pourquoi, nous n'utiliserons plus le socket comme identifiant de connexion, mais une classe. (TSocketElem)

TServerSocketElem	
Socket : TSocket	Identifiant de la Connexion (pt de vue de l'API Winsock)
DataIn : TsocketListMessageIn	Liste des messages en cours de réception.
DataOut : TSocketListMessageOut	Liste des messages en cours d'envoi.

Cette classe, contrairement à la classe TTCPClient n'est pas du tout générique. Elle est prévue pour fonctionner de manière efficiente sous certaines conditions. Deux threads suffisent au fonctionnement de ce serveur. Un s'occupant de la réception des messages et l'autre de l'envoi. Il ne faut surtout pas prévoir plusieurs threads pour faire le même Job. Cela obligerait de mettre en place toute une série de synchronisations entre ces threads et toute une série de mécanismes pour s'assurer que, lors de la reconstruction des messages, les paquets de données ne sont pas mélangés. Cela ne ferait que ralentir l'ensemble du processus.

Le sous-système réseau du middle-tier devra se plier à ce fonctionnement.

TTCPServer

Propriétés

ServerSocket : TServerSocketElem

Identifiant de point de connexion Local. (Listener)

ClientSocket : TStringListTS

Liste des TServerSocketElem identifiant chaque client connecté.
Cette liste est thread safe pour éviter des problèmes si, plus tard, des méthodes sont développées pour consulter cette liste à partir d'autre thread.

Events : TList

Liste des objets de synchronisations (Events) auxquelles seront associés les sockets

TCPInEventList : TTCPEventList

Liste des évènements fournis par la couche TCP.

TCPOutEventList : TTCPEventList

Liste des évènements fournis par la méthode WriteClient pour signifier qu'il reste encore des données à envoyer.

Méthodes

Create

Constructeur de la classe

Accept : Tsocket

Accepte une demande de connexion (appel à TTCPBase.Accept)
Ajoute la nouvelle connexion à la liste ClientSocket.
Associe le Socket Reçu à un event.

Listen (Port : Integer)

Appel TTCPBase.Listen(Port)
Associe le Socket Reçu à un event.

PollTCP (Timeout : DWord) : Boolean

Attente sur l'ensemble des events associés à des sockets (WSAWaitForMultipleEvent)
Si pas de Timeout, détection des évènements (WSAEnumNetworkEvents) et ajout de ceux-ci dans la liste TCPInEventList.
Retour :

- False : Le timeout a expiré.
- True : Des évènements ont été ajoutés à TCPEventList

ReadClient (Socket : TSocket)

- Lecture de données sur la connexion désignée par Socket. Lecture à concurrence de cReceiveBufferSize.
- Ajout du buffer lu à la liste des messages reçus associé à la connexion.(TsocketListMessageIn).

WriteClient (Socket : TSocket)

- Envoi des données présentes dans la liste des messages à envoyer pour cette connexion. Envoi à concurrence de cSendBufferSize.
- S'il reste encore des données à envoyer, on ajoute un event FD_WRITE à la liste TCPOutEventList.

Disconnect(Socket : TSocket)

Appel TTCPBase.Disconnect

DisconnectAll

Appel TTCPBase.Disconnect pour tous les sockets présents dans ClientSocket

Unlisten

Appel TTCPBase.Disconnect(ServerSocket)

GetConnectionList : Tlist

Renvoie l'instance de ClientSocket

GetConnectionInfo(Socket : TSocket ; Var ConnectionInfo : TConnectionInfo)

Appel TTCPBase.GetConnectionInfo(Socket, ConnectionInfo)

3. Service IPC (SIC)

a) Liste FIFO

Les méthodes décrites dans l'analyse fonctionnelle sont légèrement adaptées pour correspondre au paradigme orienté objet.
Les méthodes AttachFifoList et DetachFifoList sont devenues inutiles dans l'environnement technique choisi. (multi-thread et non multi-process)

TFIFO
Méthodes
Constructor Create
Création de la classe et initialisation des structures nécessaires.
Destructor Destroy
Libération de la mémoire allouée pour le fonctionnement de la classe.
Delete
La liste est entièrement effacée.
Push(ptr : Pointer; Priorité : Tpriority CF Figure)
Insertion dans la liste d'un élément référençant ptr.
La place de l'insertion est régie par deux règles :
- L'élément est placé en fin de liste sans contrer la deuxième règle.
- Un élément de priorité X sera toujours placé avant un élément de priorité X+1 ;

Pop(Var ptr : Pointer)
Place dans ptr la référence se trouvant dans le premier élément de la liste.
Cet élément est également retiré de la liste.
WaitPop(Var Ptr : Pointer; Timeout : DWORD): Boolean
Attend que la liste contienne au moins un élément.
Retour :
- **False** : Après un Timeout exprimé en ms, si aucun élément ne se trouve dans la liste.
- **True** : Un élément a été placé dans la liste endéans le timeout.
  La référence se trouvant dans le premier élément de la liste est placée dans ptr et l'élément est retiré.

Le code doit être thread safe. (Utilisation de section critique).
L'attente dans WaitPop doit être implémentée en utilisant un sémaphore reprenant le nombre d'éléments dans la liste. De cette manière, l'attente pourra se faire au moyen d'une fonction de Windows (WaitForSingleObject,…) ne prenant quasi aucune ressource CPU.

Type Tpriority = (pHigh, pNormal, pLow, pBatch)

Figure IV-1 : TPriority

4. Service Sérialisation (SSR)

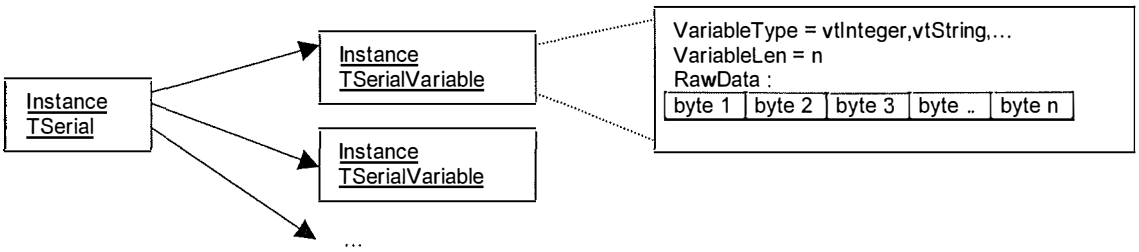
Ce service permet de gérer un ensemble de variables comme une seule entité (container)

Une fois encore, les méthodes seront adaptée pour tenir compte du paradigme objet et des fonctionnalités offertes par Delphi.

Principe de fonctionnement :

Chaque variable du container sera représentée par une instance de TSerialVariable comportant l'ensemble des méthodes nécessaires au stockage et à la récupération **d'une variable** de type quelconque.

Ces instances de classes seront centralisées par une instance de TSerial pour former le container. Celui-ci disposera des méthodes nécessaires à la sauvegarde et a la récupération des variables sous forme d'une suite d'octets. La classe TSerial dispose d'un pointeur vers chaque instance TSerialVariable qui la compose.



a) TSerialVariable

Cette classe conserve donc une valeur de type quelconque. A ce niveau, aucun nom de variable n'y est associé. C'est le container qui conserve l'association entre le nom de la variable et une instance de cette classe.

TSerialVariable
Propriétés
Type : TVType (R)
Type de la variable.

TVType = (vTSting,vtInteger,vtBoolean,vtSerial,vtStrings,vtDate, ...)

AsString : String (R/W)
Propriété permettant de stocker/lire une valeur de type String
AsInteger : Integer (R/W)
Propriété permettant de stocker/lire une valeur de type integer
AsBoolean : Boolean (R/W)
Propriété permettant de stocker/lire une valeur de type Boolean
AsSerial : TSerial (R/W)
Propriété permettant de stocker/lire une valeur de type TSerial
AsStrings : TStrings (R/W)
Propriété permettant de stocker/lire une valeur de type TStrings
AsDate : TDateTime (R/W)
Propriété permettant de stocker/lire une valeur de type Date
As...
...
Méthodes
Constructor Create
Création de la classe et initialisation des structures nécessaires.
Destructor Destroy
Libération des structures allouées.

b) TSerial

TSerial

Propriétés

Variable (Variable_Name : String) : TSerialVariable

Selon le nom de la variable, cette fonction renverra, soit :

- une nouvelle instance de TSerialVariable, si la variable n'existe pas encore.
- un instance TSerialVariable préalablement allouée si la variable existe.

Count : Integer

Retourne le nombre de variables gérées par cette classe.

Méthodes

Constructor Create

Création de la classe et initialisation des structures nécessaires.

Destructor Destroy

Libération des strucures allouées.

DeleteVariable (Variable_Name : String)

Libère l'instance TSerialVariable associée à la variable

DeleteAllVariable

Supprime l'ensemble des variable de la classe

GetVariableList (VList : TStrings)

Remplit VList avec les noms des variables contenues dans la classe

SaveToStream(Stream : TStream ; ResetStream : Boolean)

Sauve l'ensemble des variables (Nom + Valeurs associées) dans une stream.
Si ResetStream est vrai, la stream est effacée avant la sauvegarde. Sinon, les données sont ajoutées aux données existante.

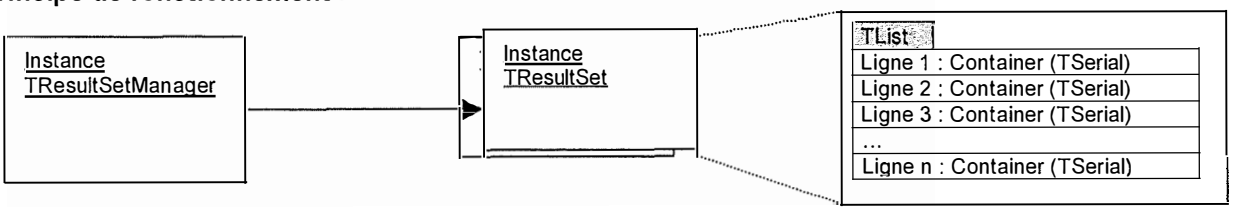
LoadFromStream(Stream : TStream ; RewindStream : Boolean)

Ajoute à la classe, les variables contenues dans la stream. Si des variables de même nom existent, celle-ci sont remplacées.
Si RewindStream est vrai, la stream est positionnée au début avant la lecture. Sinon, les données sont lues à partir de la position courante de la stream.

5. Service ResultSet (SST)

Ce service gère l'ensemble des ResultSet généré par les méthodes des OA.

Principe de fonctionnement :



Chaque ResultSet est représenté par une instance de TResultSet. L'ensemble des ResultSet est géré par une instance de TResultSetManager.
Il existe donc un seul TResultSetManager reprenant tous les TResultSet.

a) Le ResultSet

Chaque ResultSet a en charge de :

- Ajouter des lignes dans le ResultSet
- Renvoyer des lignes du ResultSet
- Rendre possible la navigation dans ce ResultSet

TResultSet
Propriétés
ID : Integer
Identifiant du ResultSet
Méthodes
Create
Constructeur de la classe
Destroy
Destructeur de la classe
SaveData(Data : TSerial)
Sauvegarde le container Data à la fin du ResultSet
GetData :TSerial
Renvoi le container se trouvant à la position courante.
Renvoi Nil si le ResultSet est vide.
First :Boolean
Positionne le ResultSet à la première ligne de données.
Renvoi False si le ResultSet est vide.
Last :Boolean
Positionne le ResultSet à la dernière ligne de données.
Renvoi False si le ResultSet est vide.
Next :Boolean
Positionne le ResultSet à la ligne suivante.
Renvoi False si il n'y a pas de ligne disponible.
Prev :Boolean
Positionne le ResultSet à la ligne précédente.
Renvoi False si le ResultSet se trouve à la première ligne.

Les données seront conservées dans une liste de type TList. La méthode SaveData ajoutera le container à la fin de la liste.

Attention : la liste contient un pointeur sur le container(TSerial) originel. En aucun cas, une copie n'est effectuée. Cela, pour des raisons de performances.
Lorsque le ResultSet est détruit, tous les containers sont détruits. La méthode de l'OA qui crée le contrainer **ne doit pas** le détruire par la suite. Le ResultSet s'en charge.

b) Le gestionnaire de ResultSet

Le gestionnaire a en charge de :

- Créer des ResultSet
- Effacer des ResultSet
- Renvoyer un pointeur sur un ResultSet existant.

Le gestionnaire est créé à l'initialisation du serveur et existera jusqu'à la clôture du serveur.

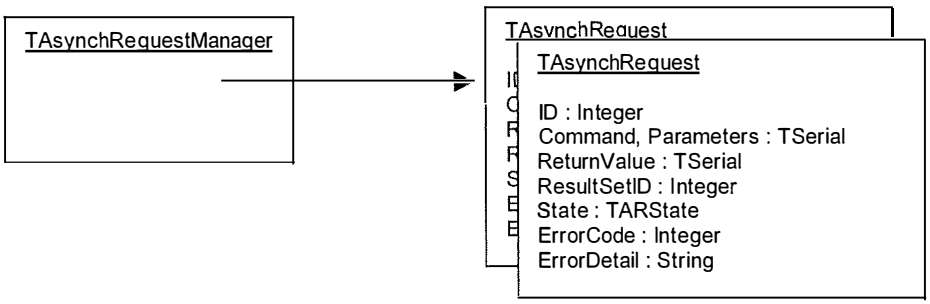
TResultSetManager
Méthodes
CreateResultSet : TResultSet
Crée un ResultSet et renvoie son identifiant
DeleteResultSet (RSID : Integer)
Supprime le ResultSet **RSID**
GetResultSet (RSID : Integer) :TResultSet
Renvoie l'instance du resultSet correspondant à RSID

La liste des ResultSet (TResultSet) maintenue au niveau du ResultSetManager est gérée par une classe TList

6. Service AsynchRequest (ASY)

Ce service gère l'ensemble des requêtes asynchrone du serveur.

Principe de fonctionnement :



Chaque demande d'exécution d'une méthode asynchrone est représentée par une instance de TAsynchRequest.
L'ensemble des TAsynchRequest est géré par une instance de TAsynchRequestManager.

a) Classe TAsynchRequest

TAsynchRequest
Propriétés

ID : Integer
Identifiant de la requête asynchrone.

Command :TSerial
Demande de l'exécution asynchrone en provenance du client.
Contient l'identifiant de session, d'OA ainsi que le nom de la méthode à exécuter.

Parameters : Tserial
Paramètres à utiliser pour l'exécution de la méthode.

ReturnValue : Tserial
Valeur de retour générée par l'exécution de la méthode.

ResultSetID : Integer
Identifiant du Resultset associé à cette requête.

State : TARState
Décrit l'état de la requête.
Cf. ci-dessous

ErrorCode : Integer
Reprend le code d'erreur éventuel généré lors de l'exécution de la méthode.

ErrorDetail : Message
Reprend le message d'erreur éventuel généré lors de l'exécution de la méthode.

Type TARState = (arNotStarted, arStarted, arTerminated)

Figure IV-2 : TARState

b) Classe TAsynchRequestManager

Le gestionnaire a en charge la création et la destruction des instances TAsynchRequest.

TAsynchRequest

Méthodes

Constructor Create

Constructeur de la classe

Destructor Destroy

Destructeur de la classe

Attention : Toutes les instances TAsynchRequest créées doivent être détruites.

CreateAsynchRequest : Integer

Crée une instance TAsynchRequest et renvoie son identifiant

DeleteAsynchRequest (ReqID : Integer)

Libère une instance TAsynchRequest précédemment créée.

7. Service objet d'application (SOA)

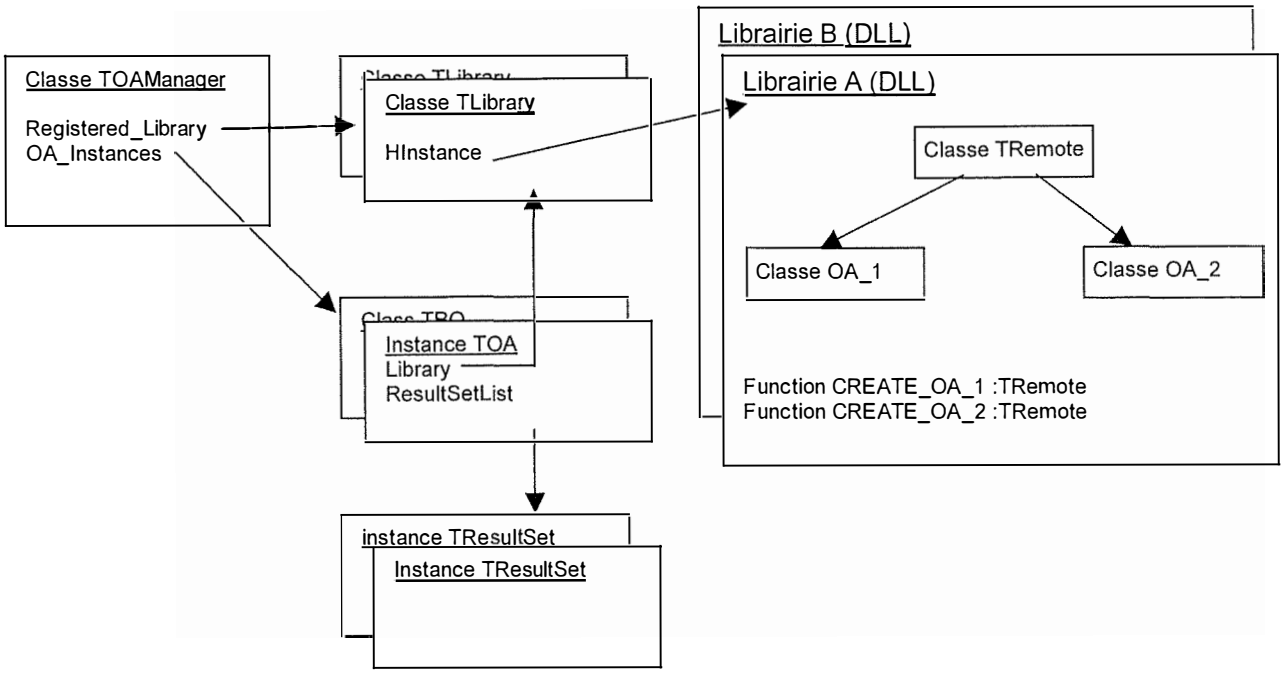
Ce service gère les objets d'applications.

principe de fonctionnement :

Tous les OA créés sont représentés par une instance de TOA. L'ensemble de ces classes est géré par une instance de TOAManager.

Lorsqu'un client demande la création d'un objet d'application (OA_CREATE), une instance de TOA est créée par le TOAManager.
Cette instance contient :

- Un pointeur sur l'instance TLibrary correspondant à la librairie d'objet d'application en question.
- Une liste de pointeur sur les TResultset généré lors de l'exécution de méthode de cet OA.



Les différentes entités que ce service doit gérer sont :

- Les librairies contenant le code des objets d'applications (TLibrary)
- Les Objets d'applications. (TOA)
- Le gestionnaire des objets d'applications (TOAManager)

a) Les librairies

Les librairies sont des dll contenant :

- des classes d'objets d'applications.
- des constructeurs d'instance de ces OA

Toutes les librairies enregistrées sont associées à une classe TLibrary. Cette classe aura en charge la gestion de la dll (Chargement / Déchargement en mémoire)

TLibrary

Propriétés

HInstance (Read)

Contient la référence de la dll chargée en mémoire. A la première utilisation de cette propriété, la dll est chargée en mémoire.

Méthodes

IncInstance

Méthode permettant d'incrémenter le nombre d'instances associées à cette librairie.

DecInstance

Méthode permettant de décrémenter le nombre d'instances associées à cette librairie. Lorsque le nombre d'instance est égale à 0, la dll est déchargée de la mémoire

Les classes des objets d'applications

Ces classes sont des classes normales, sauf qu'elles doivent respecter certaines règles :

- Dériver d'une même classe de base (**TRemote**) comprenant l'ensemble des méthodes nécessaires à une exécution dynamique.
- Le constructeur de la classe doit respecter la signature suivante :
  Constructor Create (Parameters : TSerial)
- Les méthodes destinées à être exécutées à distance doivent respecter la signature suivante :
  Procedure MethodName(ParamIn : TSerial; Response : TSerial; ResultSet : TResultSet)
- Les méthodes destinées à être exécutées à distance doivent êtres placées dans la section published de la classe. (Utilisation des info RTTI)

TRemote

Méthodes

Execute(method: string;Parameters : TSerial;ReturnValue : TSerial ; ResultSet : TResultSet)

Exécute une méthode de la classe(*), passe parameters, contenant les paramètres de la méthode, passe ReturnValue destiné à recevoir le résultat de l'exécution. Passe aussi ResultSet qui permettra à la méthode de générer un ResultSet

(*) L'exécution dynamique d'une méthode d'une classe peut se faire très facilement en utilisant le RTTI (Runtime Type Information) du Delphi :

```
TRemoteProc = Procedure(ParamIn : TSerial; ReturnValue : TSerial; ResultSet : TResultSet) of object;

Procedure TRemote.execute(method: string;Param : TSerial;ReturnValue : TSerial ; ResultSet : TResultSet);
Var m : TRemoteProc;
begin
 TMethod(M).Code := Self.MethodAddress(method);
 If TMethod(M).Code = Nil Then Raise ERSOException.Create(ERR_INVALID_METHOD,Method);
 TMethod(M).Data := Self;
 M(Param,Response,ResultSet);
end;
```

Les constructeurs d'instance d'OA

Ceci est un cas particulier, on ne peut créer une instance de la même façon que l'on exécute une méthode.
Il est nécessaire d'utiliser un subterfuge en ce qui concerne la création d'un objet d'application.
La librairie doit implémenter une fonction qui créera une instance de l'objet d'application voulu.
Cette fonction aura pour nom, une structure précise, qui permettra au OAManager d'exécuter celle-ci au moment voulu.
Signature: **Procedure CREATE_NOMCLASSE (Var Inst : TREMOTE; Parameters : TSerial);**

L'instance ainsi créée sera renvoyée dans **Inst**. Les paramètres du constructeur de la classe seront passés dans **Parameters**.

b) Les Objets d'applications

Chaque instance d'objet d'application créé par un client est représenté au niveau du serveur par une classe TOA. Cette classe va conserver les informations suivantes :

- L'adresse de l'instance.
- La librairie dont elle est issue
- La liste des ResultSet appartenant à cette instance.

Celle-ci a donc en charge la gestion des ResultSet associés à cet OA.

TOA
Propriétés
ID : Integer
Identifiant de l'OA
Instance : TRemote
Instance de l'OA
LibraryName : String
Nom de la librairie dont est issue l'instance
Méthodes
Create(ResultSetManager : TResultSetManager)
Constructeur de la classe.
L'instance du ResultSetManager est passée pour permettre la gestion des différents ResultSet de cet OA
Destroy
Destructeur de la classe.
L'ensemble des ResultSet encore actifs est détruit.
AddRS:TResultSet
Crée un ResultSet pour cet OA
DeleteRS (RSID : Integer)
Supprime le ResultSet spécifié.
ExistRS (RSID : Integer)
Teste l'existence du ResultSet Spécifié.
CountRS : Integer
Retourne le nombre de ResultSet associé à cet OA.
GetListRS (List : TStrings)
Remplit la liste avec les identifiants de ResultSet associé à cet OA.

c) Le gestionnaire des objets d'applications

Ce gestionnaire va gérer l'ensemble des OA créés pour le serveur d'application. Il ne peut en exister qu'un.

Ses tâches sont les suivantes:

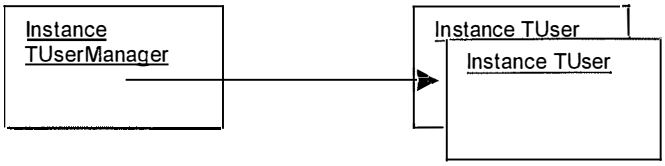
- Enregistrement des librairies accessibles par les clients
- Création d'instance d'OA
- Exécution de méthodes d'OA
- Libération des instances précédemment créées

TOAManager
Méthodes
Create(ResultSetManager : TResultSetManager)
Constructeur de la classe.
L'instance du ResultSetManager est passée pour pouvoir le transmettre aux classes TOA
Destroy
Destructeur de la classe
RegisterLibrary (LibraryName : String)
Crée une classe TLibrary pour la librairie passée en paramétre
CreateInstance (Lib : String; OA : String; Param : TSerial) : Integer
Crée une instance de la classe OA se trouvant dans Lib et passe Param au constructeur de la classe.
L'identifiant de l'OA créé est renvoyé.
ExecMethod(OAID : Integer; Method : String; Param : TSerial; Response : TSerial ; Var ResultSet : TResultSet)
Exécute la méthode Method de l'OA OAID, passe Param, Response et ResultSet à la méthode.
FreeInstance(OAID : Integer)
Libère l'instance OAID
GetOA(OAID : Integer)
Retourne l'instance de TOA associée à OAID

8. Service Sécurité (SSC)

Ce service gère l'ensemble des utilisateurs connus du système.

Principe de fonctionnement :



Chaque utilisateur est représenté par une classe TUser. L'ensemble des utilisateurs est géré par un gestionnaire (TUserManager).

a) L'utilisateur

Chaque TUser a en charge :

- la conservation des informations propres à un utilisateur

TUser
Propriétés
UserID : String (Read)
Identifiant de cet utilisateur.
Cette propriété ne peut être modifiée puisque c'est la clé d'accès de cet utilisateur
PWD : String (Write)
Mot de passe de cet utilisateur
Name : String (Read / Write)
Nom de cet utilisateur
Desc : String (Read / Write)
Description de cet utilisateur
State : TUserState (Read/Write)
Etat de cet utilisateur
TUserState = (usActive,usNotActive)

Cette propriété permet de conserver un utilisateur tout en le désactivant temporairement.
Priority : TUserPriority (Read / Write)
Priorité de cet utilisateur
TUserPriority = (upHigh,upNormal,upLow)

Cette propriété permettra par la suite, de rendre l'exécution de certaine méthode prioritaire par rapport à d'autres en fonction de l'utilisateur qui les exécute.

Méthodes

Create(UserID,PWD,Name,Desc : String; Priority : Integer)

Constructeur de la classe.
Les caractéristiques de l'utilisateur sont sauvegardées dans un container.

Create(Stream : TStream)

Constructeur de la classe.
Les caractéristiques de l'utilisateur sont sauvegardées dans un container. Les caractéristiques de l'utilisateur sont chargées à partir de **Stream**

Destroy

Destructeur de la classe.
Le container conservant les caractéristiques est détruit.

Grant(Lib : String)

Donne le droit à cet utilisateur de créer des OA contenus dans la librairie **Lib.**

Revoke(Lib : String)

Retire le droit à cet utilisateur de créer des OA contenus dans la librairie **Lib.**

SaveToStream (Stream : Stream)

Les caractéristiques de l'utilisateur sont sauvegardées dans **Stream**

checkPassword (PWD : String)

Vérifie que le mot de passe de l'utilisateur correspond à celui passé en paramètre.

b) Le gestionnaire des utilisateurs

Le gestionnaire a en charge :

- l'ajout, la modification et la suppression d'utilisateurs.
- la sauvegarde permanente des informations.

TUserManager

Méthodes

AddUser(UserID,PWD,Name,Desc : String; Priority : Integer)

Crée un nouvel utilisateur
Remarque : le UserID est fourni par une couche supérieure. Il doit être unique. Si un même UserID existe déjà, l'ajout doit se solder par un échec.

DeleteUser(UID : String)

Supprime un utilisateur

GetUser(UID : String):TUser;

Renvoie un utilisateur sur base de son identifiant

GetUserList(List : TStringList)

Remplit **List** avec la liste des identifiants des utilisateurs connus du système

SaveUsers(Filename : String)

Sauvegarde les utilisateurs dans le fichier spécifié par Filename

ReadUsers(Filename : String)

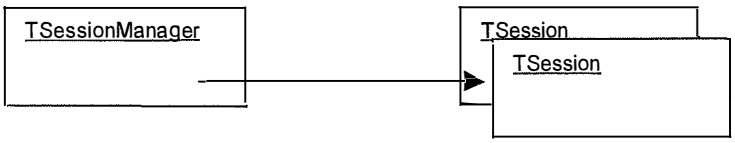
Lit les utilisateurs contenus dans le fichier spécifié par Filename

Les TUser sont maintenus au sein du UserManager par une classe de type TList.

9. Service Session (SSM)

Ce service gère les sessions au niveau du serveur.

Principe de fonctionnement



Chaque session est représentée par une classe TSession. L'ensemble des sessions est géré par un gestionnaire (TSessionManager).

a) La session

La classe TSession a en charge :

- le maintien des caractéristiques d'une session

TSession
Propriétés
ID : Integer
Identifiant de la session
UserID : String
Identifiant de l'utilisateur qui possède cette session
CreationDate : TDateTime
Date de création de la session
State : TSessionState
Etat de la session
TSessionState =(ssConnected,ssDisconnected)

Méthodes
Create (ID : Integer ; UserID : String)
Constructeur de la session
Destroy
Destructeur de la session
AddOA(OAID : Integer)
Ajoute l'identifiant de l'OA à la liste des OA appartenant à cette session.
DeleteOA(OAID : Integer)
Supprime l'identifiant de l'OA de la liste des OA appartenant à cette session.
ExistOA(OAID : Integer) : Boolean
Renvoi True si l'OA appartient à la session en question
GetListOA(List : TStringList)
Remplit List avec la liste des identifiants d'OA appartenant à cette session

b) Le gestionnaire de session

Le gestionnaire a en charge :

- La création et la suppression de session

TSessionManager

Méthodes

Create

Constructeur de la classe

Destroy

Destructeur de la classe

CreateSession(UserID : String):Integer

Crée une session (TSession) et renvoi l'identifiant de cette session

DeleteSession(SEID : Integer)

Supprime la session spécifiée

GetSession(SEID : Integer)

Renvoi l'instance de TSession associée à **SEID**

ExistSession(SEID : Integer) :Boolean

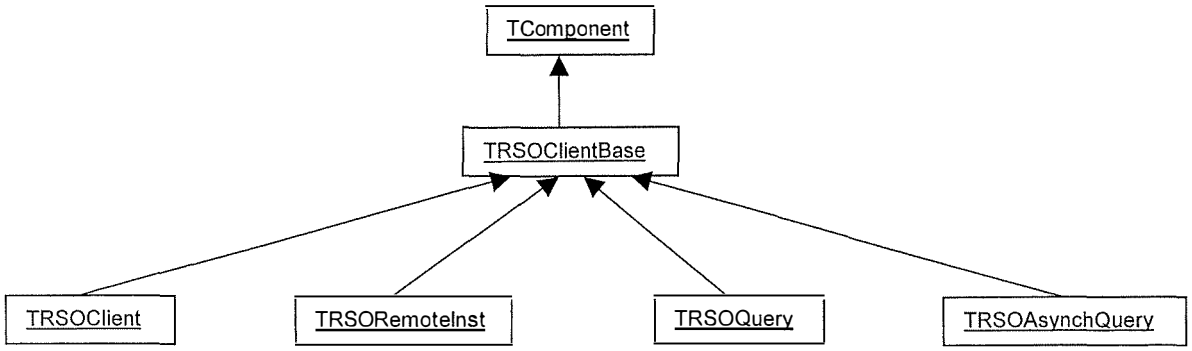
Renvoi True si l'identifiant de session existe.

La liste des sessions (TSession) est maintenue au niveau du SessionManager par une classe TList. L'identifiant est un nombre aléatoire garanti unique pour toutes les sessions

E. Analyse des modules

1. Module client

Pour pouvoir tirer parti des avantages de Delphi et pour faciliter le développement d'un client, l'ensemble des fonctionnalités nécessaires va être découpé en classes ayant des tâches bien précises.
Ces classes seront dérivées de TComponent. En faisant de la sorte, nous pourront développer des clients de façon visuelle, en « déposant » des composants sur les fenêtres.



TRSOClientBase :	Classe de base commune à l'ensemble des composants RSO client
TRSOClient :	Gère une connexion vers un serveur ~ TSession
TRSORemoteInst :	Gère une instance d'un OA
TRSOQuery :	Gère l'exécution synchrone d'une méthode et le ResultSet éventuel associé.
TRSOAsynchQuery :	Gère l'exécution asynchrone d'une méthode et le ResultSet éventuel associé.

a) TRSOClientBase

Chaque classe a besoin de stocker temporairement le message à envoyer au serveur ainsi que la réponse reçue de celui-ci.
Ce stockage se fait par l'intermédiaire d'une classe TMemoryStream. Cette classe est très pratique, elle permet de stocker en mémoire une suite de byte de longueur variable.

De plus, si par la suite, le besoin se fait sentir d'ajouter des propriétés communes à toutes les classes, la mise à jour sera très simple.

TSessionManager
Propriétés
RawMessage : TMemoryStream
Contient, soit le message courant, soit, la réponse courante
Méthodes
Create(AOwner : TComponent)
Constructeur de la classe.
Crée le TMemoryStream
Destroy
Destructeur de la classe.
Libère le TMemoryStream

b) TRSOClient

Cette classe a pour tâche de :

- Gérer la connexion physique avec le serveur.
- Gérer une session. (Création / attachement / déconnexion).

TRSOClient

Propriétés

TCPClient : TTCPClient

Instance d'une classe TTCPClient en charge du réseau

SessionID : Integer

Identifiant de la session remote. (-1 si pas encore créée)

Host : String

Adresse TCP ou Nom du serveur RSO

Port : Integer

Port du serveur RSO

UserID : String

Identifiant de l'utilisateur

Password : String

Mot de passe de l'utilisateur

Méthodes

Create(AOwner : TComponent)

Constructeur de la classe.

Destroy

Destructeur de la classe.

CreateSession

Crée une session sur le serveur.

```
TCPClient.Connect (Host :Port)
Send  (CMD = SESSION_CONNECT ; SESSION = -1 ; UID = UserID ; PWD = Password)
Wait&Load InternalResponse
if InternalResponse.CMD = SESSION_CONNECT_OK
        SessionID= InternalResponse.Session
if InternalResponse.SESSION_CONNECT_KO
        SessionID = -1
        Call ErrorManagement
```

AttachSession

S'attache à une session existant sur le serveur.

```
TCPClient.Connect(Host,Port)
Send (CMD = SESSION_CONNECT ; SESSION = SessionID ; UID = UserID ; PWD =
Password)
Wait&Load InternalResponse
if InternalResponse.CMD = SESSION_CONNECT_OK
        SessionID= InternalResponse.Session
if InternalResponse.CMD = SESSION_CONNECT_KO
        SessionID = -1
        Call ErrorManagement
```

Disconnect(DisconnectMode : TDisconnectMode)

Se déconnecte du serveur.

```
Send (CMD = SESSION_DISCONNECT ; SESSION = SessionID ; MODE = DisconnectMode)
Wait&Load InternalResponse
if InternalResponse.CMD = SESSION_DISCONNECT_OK
        SessionID = -1
        TTCClient.Disconnect
if InternalResponse.CMD = SESSION_DISCONNECT_KO
        SessionID = -1
        Call ErrorManagement
```

c) TRSORemoteInst

Cette classe a pour tâche de :

- Créer / libérer des instances d'OA remote.

TRSORemoteInst

Propriétés

RSOClient : TRSOClient

Instance RSOClient à laquelle cette classe se rapporte.

OAID : Integer

Identifiant de l'OA distant.

Param : TSerial

Paramètre utilisé lors de la création de l'instance

Méthodes

Create(AOwner : TComponent)

Constructeur de la classe

Destroy

Destructeur de la classe

OACreate(Lib : String; OA : String)

Crée une instance de la classe OA appartenant à la librairie Lib sur le serveur

```
Send (CMD = OA_CREATE ; LIB = Lib ; OA = OA; Session = RSOClient.SessionID) || Param
Wait&Load InternalResponse
if InternalResponse .CMD = OA_CREATE_OK
        OAID = InternalResponse.OA
if InternalResponse .CMD = OA_CREATE_KO :
        OAID = -1
        Call errorManagment
```

OAFree

Libère l'instance courante sur le serveur

```
Send (CMD =  OA_FREE ; OA = OAID ; Session = RSOClient.SessionID)
Wait&Load InternalResponse
if InternalResponse.CMD = OA_FREE_OK
        OAID = -1
if InternalResponse.CMD =OA_FREE_KO
        OAID = -1
        Call ErrorManagement
```

d) TRSOQuery

Cette classe a pour tâche de :

- Exécuter une méthode distante de façon synchrone.
- Gérer le ResultSet éventuel.

TRSOQuery
Propriétés
RSORemoteInst : TRSORemoteInst
Instance RSORemoteInst à laquelle cette classe se rapporte.
Parameters : TSerial
Paramètres de la méthode
Response: TSerial
Réponse de l'exécution de la méthode.
ResultSetID: integer
Identifiant du ResultSet éventuel
ResultSet: TSerial
Contenu de la ligne courante du ResultSet
EOF : Boolean
Indique la fin du ResultSet
Méthodes
Create(AOwner : TComponent)
Constructeur de la classe
Destroy
Destructeur de la classe
Execute(Method : String)
Exécute la méthode distante Method de façon synchrone et positionne le resultSet(s'il existe) sur la première ligne.

```
Send (CMD = OA_EXEC ; METHOD = Method ; OAID = RSORemoteInst.OAID ; SESSION =
RSORemoteInst.RSOClient.SessionID) || Parameters
Wait&Load InternalResponse
if InternalResponse.CMD = OA_EXEC_OK
        ResultSetID= InternalResponse.RESULTSET
        Load Response
        call ResultSetFirst
if InternalResponse.CMD = OA_EXEC_KO
        ResultSetID=-1
        Call ErrorManagement
```

ResultSetFirst,
ResultSetLast,
ResultSetNext,
ResultSetPrev
Positionne le ResultSet sur la première ligne.

```
Send (CMD = RESULTSET_FIRST ; RESULTSETID=ResultSetID)
Wait&Load InternalResponse
if InternalResponse.CMD = RESULTSET_FIRST_OK
        EOF= InternalResponse.EOF
        Load ResultSet
if InternalResponse.CMD = RESULTSET_FIRST_KO
        Call ErrorManagement
```

e) TRSOAsynchQuery

Cette classe a pour tâche de :

- Exécuter une méthode distante de façon asynchrone
- Gérer le ResultSet éventuel

TRSOAsynchQuery

Propriétés

RSORemoteInst : TRSORemoteInst
Instance RSORemoteInst à laquelle cette classe se rapporte.

Parameters : TSerial
Paramètre de la méthode.

ReqID: Integer
ID de la requête en cours.

Response: TSerial
Réponse de l'exécution de la méthode.

ResultSetID: integer
Identifiant du ResultSet éventuel.

ResultSet: TSerial
Contenu de la ligne courante du ResultSet.

EOF : Boolean
Indique la fin du ResultSet.

State : TAsynchQueryState
Etat du composant. Soit, aucune méthode n'est encours d'exécution, soit une méthode est en cours, soit l'exécution est terminée et le résultat disponible.
TAsynchQueryState = (aqsIdle,aqsExecuting,aqsTerminated)

PollInterval : Integer
Temps en secondes entre chaque appel automatique à CheckState.
Si PollInterval=0, aucun appel automatique n'est effectué.

Méthodes

Create(AOwner : TComponent)
Constructeur de la classe
State = aqsIdle

Destroy
Destructeur de la classe

Execute(Method : String)
Exécute la méthode distante Method de façon asynchrone.

```
State = aqsExecuting
Send (CMD = OA_EXEC_ASYNCH ; METHOD = Method  ; OAID = RSORemoteInst.OAID ;
SESSION = RSORemoteInst.RSOClient.SessionID) || Parameters
Wait&Load InternalResponse
if InternalResponse.CMD = OA_EXEC_ASYNCH_OK
        ReqID = internalResponse.ReqID
        ResultSetID = InternalResponse.RESULTSETID
if InternalResponse.CMD = OA_EXEC_ASYNCH_KO
        ResultSetID = -1
        ReqID = -1
        State = aqsIdle
        Call errorManagement
```

ResultSetFirst,
ResultSetLast,
ResultSetNext,
ResultSetPrev

Positionne le ResultSet sur la première ligne.

```
Send (CMD = RESULTSET_FIRST ; RESULTSETID = ResultSetID ; SESSION =
RSORemoteInst.RSOClient.SessionID)
Wait&Load InternalResponse
if InternalResponse.RESULTSET_FIRST_OK
        EOF = InternalResponse.EOF
        Load ResultSet
if InternalResponse.RESULTSET_FIRST_KO
        Call ErrorManagement
```

CheckState

Vérifie, au près du serveur, la terminaison de la méthode en cours d'exécution. Si celle-ci est terminée, Response est chargé.

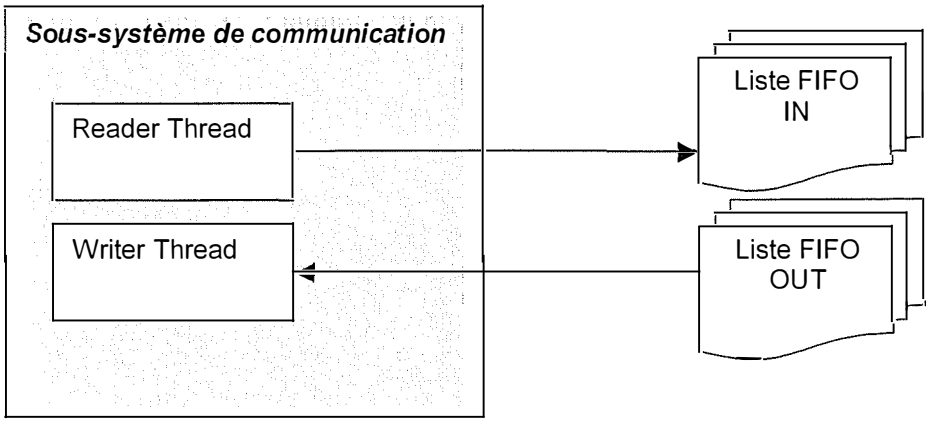
```
Send (CMD = OA_EXEC_ASYNCH_RESULT ; REQID = ReqID ; SESSION =
RSORemoteInst.RSOClient.SessionID)
Wait&Load InternalResponse
if InternalResponse.CMD = OA_EXEC_ASYNCH_RESULT_OK
        Load Response
        State = aqsTerminated
if InternalResponse.CMD =OA_EXEC_ASYNCH_RESULT_KO
        if error <> NOT_TERMINATED
                call ErrorManagement
```

2. Module Serveur

a) Sous-système de gestion des communications.

La gestion des communications fait usage de la classe TTCPServer du service Réseau.
Lors de l'analyse technique du service réseau, certains choix ont étés pris. Ces choix imposent de n'avoir qu'un seul thread s'occupant de la réception des messages. (cf. Envoi et réception des messages. Page 89)

Le fonctionnement du sous-système de communication sera donc le suivant :



Le grand principe de la couche réseau est de ne pas forcer le système. Des données sont passées à la couche winsock, celle-ci envoie ce qu'elle peut. Notre couche réseau retransmettra plus tard les données non envoyées. Ce principe est le même dans le cas de la réception.

Le « Reader Thread » s'occupe de lire des petits morceaux de messages (gérés par la classe TTCPServeur) et, lorsqu'un message est complet, le place dans Liste FIFO In.

De même, le « Writer Thread » s'occupe d'envoyer les messages présents dans Liste FIFO Out. Ceux-ci seront aussi découpés en morceaux, selon les ressources systèmes.

Pour permettre l'envoi des réponses correspondant aux messages reçus, il est nécessaire de garder trace de la provenance de ce message. Cette provenance ne devra pas être altérée par les autres sous-systèmes, afin que, le moment venu, la réponse puisse être renvoyée au client de droit.
Le message sera donc placé dans une structure TRSOMessage qui contiendra aussi l'identifiant de la connexion réseau.

TRSOMessage
Propriétés
Client : TServerSocketElem
Identifiant de la connexion physique.
Message : TmemoryStream
Message en provenance ou à destination d'un client.

TWriterThread
Propriétés
DataOut : TFifo
Liste FIFO Out
TCPServer : TTCPServer
TTCPServer (unique pour tout le serveur)
Méthodes
Create(TCPServer : TTCPServer; DataOut : TFifo)
Constructeur de la classe.
Le serveur TCP et la liste FIFO de sortie sont passés en paramètres.

Destroy

Destructeur de la classe.

Execute

Méthode principale d'un thread.

```
Repeat
Wait on DataOut & TCPServer.TCPOutEventList
If message in DataOut
        DataOut.Pop(RSOMessage)
         RSOMessage.Client.MessageOut.AddMessage(RSOMessage.Message)
        TCPServer.WriteClient(RSOMessage.Client.Socket)
If Event in TCPServer.TCPOutEventList
        TCPServer.TCPOutEventList.getEvent(Client, Event)
        If Event =FD_WRITE
                TCPServer.WriteClient(Client.Socket)
Until Terminated
```

TReaderThread

Propriétés

DataIn : TFifo

Liste FIFO In

TCPServer : TTCPServer

TTCPServer (unique pour tout le serveur)

Méthodes

Create(TCPServer : TTCPServer; DataIn : TFifo)

Constructeur de la classe.
Le serveur TCP et la liste FIFO d'entrée sont passés en paramètres.

Destroy

Destructeur de la classe.

Execute

Méthode principale d'un thread.

```
TCPServer.Listen (ServerParameters.Port[1])
Repeat
If TCPServer.PollTCP(ServerParameters.PollingTime)
        TCPServer.TCPInEventList.getEvent(Client,Event)
        Case Event of
         FD_ACCEPT :
                TCPServer.Accept
         FD_READ :
                TCPServer.ReadClient(Client.Socket)
                Message := Client.MessageIn.getData
                If Message <> Nil
                        RSOMessage.Client := Client
                        RSOMessage.Message := Message
                        DataIn.Push(RSOMessage)
        FD_CLOSE :
                TCPServer.Disconnect(Client.Socket)
Until Terminated
```

[1] L'objet ServerParameters est décrit dans le sous-système de gestion des resources, page 127.

b) Sous-système de traitement des requêtes

Ce sous-système va :

- Lire les messages de la liste FIFO In.
- Décoder le Message.
- Exécuter les actions nécessaires à la réalisation des demandes des clients.
- Placer les réponses dans la liste FIFO Out.

Ce système risque d'être soumis à une charge importante. C'est pourquoi son architecture est très importante.

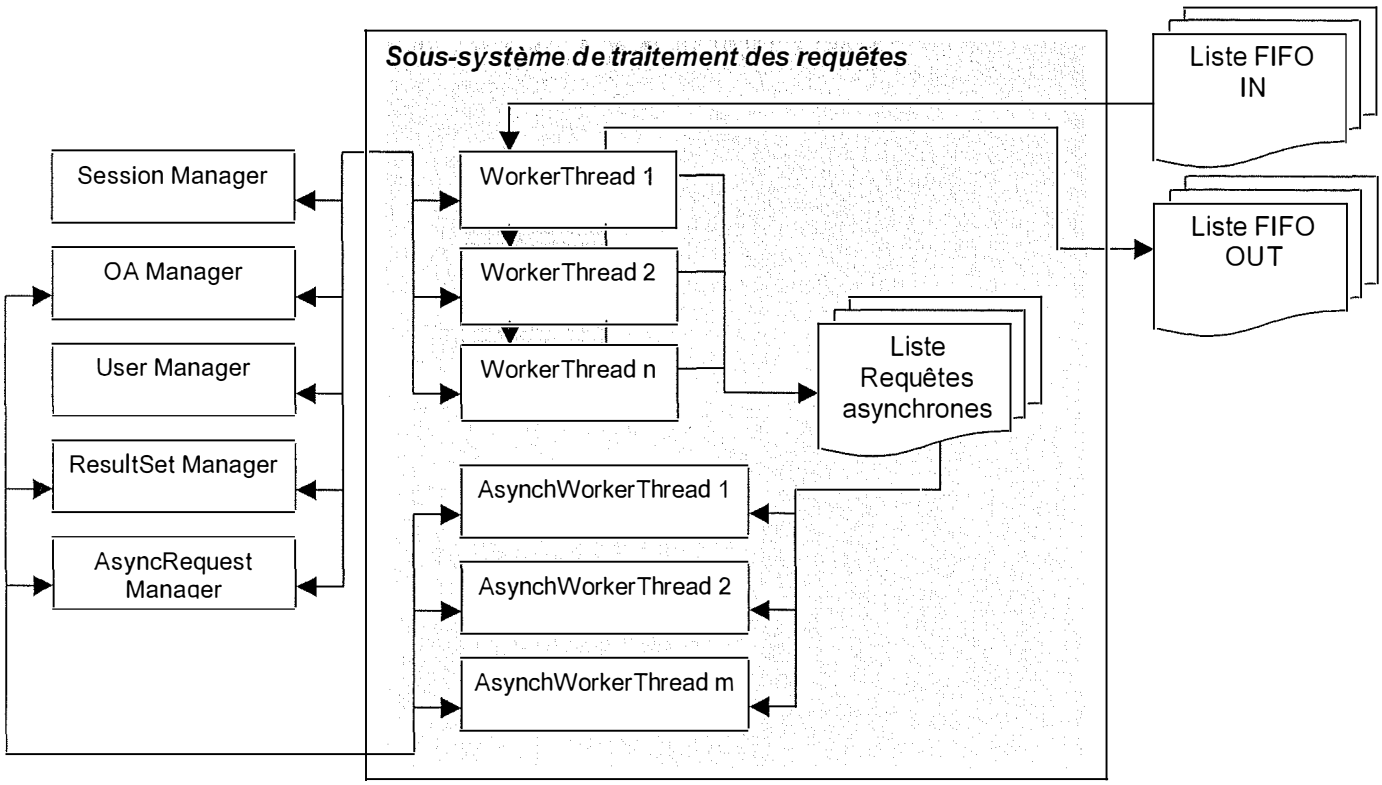
On peut distinguer trois parties distinctes :

- La gestion des requêtes des clients.
- L'exécution des méthodes synchrones
- L'exécution des méthodes asynchrones

La gestion des requêtes n'étant pas quelque chose de lourd, l'on peut sans problème la regrouper avec l'exécution des requêtes synchrone.

Il reste donc deux systèmes à résoudre. Ces deux systèmes auront chacun un certain nombre de threads à leur disposition pour réaliser le travail. Le nombre de thread doit pouvoir être adapté dynamiquement par le sous-système de gestion des ressources.

Schéma de fonctionnement du sous-système de traitement des requêtes :



-1- Gestion des requêtes & exécution synchrone

Chaque WorkerThread va récupérer les messages de Liste FIFO In et les traiter dans leur entierté. Tout message passera obligatoirement par un WorkerThread, même une demande d'exécution d'une méthode asynchrone.

Attention :

1. Chaque Message envoyé doit comporter l'identifiant de la session. Pour des raisons de clarté, cet aspect a été retiré du pseudo-code.
2. Le pseudo-code ne prévoit pas de vérification de la validité des identifiants venant des clients.

TWorkerThread
Propriétés
DataIn : Tfifo
Liste FIFO In
DataOut : Tfifo
Liste FIFO Out
AsynchRequest : Tfifo
Liste FIFO pour les requêtes asynchrone
SessionManager : TSessionManager
SessionManager (unique pour tout le serveur)
UserManager : TUserManager
UserManager (unique pour tout le serveur)
OAManager : TOAManager
OAManager (unique pour tout le serveur)
ResultSetManager : TResultSetManager
ResultSetManager (unique pour tout le serveur)
AsynchResponseManager : TasynchResponseManager
AsynchResponseManager (unique pour tout le serveur)

Méthodes

Create (DataIn, DataOut, AsynchRequest : Tfifo ; SessionManager : TSessionManager;UserManager : TuserManager ; OAManager : TOAManager; ResultSetManager : TResultSetManager);

Constructeur de la classe.
Les instances des managers sont passées au thread.

Destroy

Destructeur de la classe.

Execute

Méthode principale du thread.

```
If DataIn.WaitPop(RSOMessage, ServerParameters.PollingTime[1])
    Command, Response,Parameters,ReturnValue : TSerial
    Load Command from RSOMessage
    case Command.Cmd
        SESSION_CONNECT :
            Call SessionConnect(Command,Response)
            Save Response to ResponseMessage
        SESSION_DISCONNECT :
            Call SessionDisconnect(Command,Response)
            Save Response to ResponseMessage
        OA_CREATE :
            Load Parameters from RSOMessage
            OACreate(Command,Parameters,Response)
            Save Response to ResponseMessage
        OA_FREE :
            OAFree(Command,Response)
            Save Response to ResponseMessage
        OA_EXEC :
            Load Parameters from RSOMessage
            OAExec(Command, Parameters, Response, ReturnValue)
            Save Response, ReturnValue to ResponseMessage
        OA_EXEC_ASYNCH :
            Rewind RSOMessage
            OAExecAsynch(RSOMessage, Response)
            Save Response to ResponseMessage
        OA_EXEC_ASYNCH_RESULT :
            OAExecAsynchResult(Command, Response, ReturnValue)
            Save Response,ReturnValue to ResponseMessage
        RESULTSET_GET :
            ResultSetGet(Command,Response,ReturnValue)
            Save Response, ReturnValue to ResponseMessage
        RESULTSET_NEXT :
            ResultSetNext(Command,Response, ReturnValue)
            Save Response, ReturnValue to ResponseMessage
        RESULTSET_PREV : Begin
            ResultSetPrev(Command,Response, ReturnValue)
            Save Response, ReturnValue to ResponseMessage
        RESULTSET_FIRST :
            ResultSetFirst(Command,Response, ReturnValue)
            Save Response, ReturnValue to ResponseMessage
        RESULTSET_LAST ;
            ResultSetLast(Command,Response, ReturnValue)
            Save Response, ReturnValue to ResponseMessage
        RESULTSET_COUNT :
            ResultSetCount(Command,Response);
            Save Response, ReturnValue to ResponseMessage
        RESULTSET_FREE ;
            ResultSetFree(Command,Response);
            Save Response, ReturnValue to ResponseMessage
    DataOut.Push(ResponseMessage)
```

[1] L'objet ServerParameters est décrit dans le sous-système de gestion des resources, page 127.

SessionConnect(Command, Response : TSerial)

Etablit une nouvelle session ou s'attache à une existante.

```
If Command.session = - 1
        Response.Cmd = SESSION_CONNECT_OK
        Response.Ses = SessionManager.CreateSession
Else
        Session := SessionManager.getSession(Command.Ses)
        If Session = -1
                Response.Cmd = SESSION_CONNECT_KO
                Response.ErrorCode = ERR_SESSION_NOTEXIST
                Response.Ses = -1
        Else
                Response.Cmd = SESSION_CONNECT_OK
                Response.Ses = Session
```

SessionDisconnect (Command,Response : Tserial)

Déconnexion d'une session (destruction) ou détachement selon le mode passé en paramètre.

```
Session = SessionManager.getSession(Command.Ses))
Case Command.Mode
        dmDisconnect :
                If Session.CountOA = 0
                        SessionManager.Delete(Command.Ses)
                        Session.State = ssDisconnected
                        Response.Cmd = SESSION_DISCONNECT_OK
                Else
                        Response.Cmd = SESSION_DISCONNECT_KO
                        Response.ErrorCode = ERR_OA_ACTIVE
        dmDisconnectForce :
                Destroy All OA associated to the session
                If Session.CountOA = 0
                        SessionManager.Delete(Command.SES)
                        Session.State = ssDisconnected
                        Response.CMD = SESSION_DISCONNECT_OK
                Else
                        Response.Cmd = SESSION_DISCONNECT_KO
                        Response.ErrorCode = ERR_INTERNAL
        dmDetach :
                Session.State = ssDisconnected
                Response.CMD = SESSION_DISCONNECT_OK
```

OACreate(Command,Parameter,Response : TSerial)

Création d'une instance d'un OA

```
OAID = OAManager.CreateInstance(Command.Lib,Command.OA,Parameter)
Session.AddOA(OAID)
Response.Cmd = OA_CREATE_OK
```

OAFree(Command, Response: TSerial)

Libération d'une instance d'OA.

```
Session.DeleteOA(Command.OAID)
OAManager.Freeinstance(Command.OAID)
Response.CMD = OA_FREE_OK
```

OAExec(Command, Parameters, Response, ReturnValue : Tserial)

Exécution d'une méthode d'un OA

```
ResultSet = 0
If OAManager.ExecMethod(Command.OAID, Command.Method, Parameters, ReturnValue,
ResultSet)
        If ResultSet = 0
                Response.ResultSetID = -1
        Else
                Response.ResultSetID = ResultSet
        Response.Cmd = OA_EXEC_OK
Else
        Response.Cmd = OA_EXEC_KO
        Response.ErrorCode = ERR_OA_EXECUTE)
        Response.ErrorDetail = Exception.Message
```

OAExecAsynch(RSOMessage : TMemoryStream, Response : Tserial)

Demande d'exécution asynchrone d'une méthode

```
AR = AsynchRequestManager.CreateRequest
AR.Request = RSOMessage
RS = ResultSetManager.CreateResultSet
AR.ResultSetID = RS.ID
AsynchRequest.Push(AR.ID)
Response.Cmd=OAEXEC_ASYNCH_OK
Response.ReqID = AR.ID
Response.ResultSetID = RS.ID
```

OAExecAsynchResult(Command, Response, ReturnValue)

Renvoi le résultat(s'il existe) de l'exécution d'une requête asynchrone.

```
OA = OAManager.getOA(Command.OAID)
If OA.ExistAsynchRequest(Command.REQID)
        AR = AsynchResponseManager.get(Command.ReqID)
        If AR.State = arTerminated
                If AR.ErrorCode = ERR_NONE
                        ReturnValue = AR.ReturnValue
                        Response.Cmd := OA_EXEC_ASYNCH_RESULT_OK
                Else
                        Response.Cmd := OA_EXEC_ASYNCH_RESULT_KO
                        Response.ErrorCode := AR.ErrorCode
                        Response.ErrorDetail := AR.ErrorDetail
                AsynchRequestManager.Delete(Command.ReqID)
        Else
                Response.Cmd = OA_EXEC_ASYNCH_RESULT_KO
                Response.ErrorCode = ERR_OAASYNCH_NOTTERMINATED
Else
        Response.Cmd = OA_EXEC_ASYNCH_RESULT_KO
        Response.ErrorCode = ERR_OAASYNCH_INVALID_REQID)
```

ResultSetGet(Command, Response, ReturnValue : TSerial)

Renvoie la ligne courante d'un ResultSet

```
OA = OAManager.getOA(Command.OAID)
If OA.ExistRS(Command.RSID)
        RS = ResultSetManager.getResultSet(Command.RSID)
        ReturnValue = RS.getData
        Response.Cmd = RESULTSET_GET_OK
Else
        Response.Cmd = RESULTSET_GET_KO
        Response.ErrorCode = ERR_RESULTSET_INVALID
```

ResultSetFirst(Command,Response, ReturnValue : TSerial)
ResultSetLast(Command,Response, ReturnValue : TSerial)
ResultSetNext(Command,Response, ReturnValue : TSerial)
ResultSetPrev(Command,Response, ReturnValue : TSerial)
Déplace un ResultSet et renvoi la ligne courante de celui-ci.

```
OA = OAManager.getOA(Command.OAID)
If OA.ExistRS(Command.RSID)
        RS = ResultSetManager.getResultSet(Command.RSID)
        If RS.First || RS.Last || RS.Next || RS.Prev
                Response.EOF = 0
                ReturnValue = RS.getData
                Response.Cmd = RESULTSET_FIRST_OK
        Else
                Response.EOF = 1
                ReturnValue = 0
                Response.Cmd = RESULTSET_FIRST_OK
Else
        Response.Cmd = RESULTSET_FIRST_KO
        Response.ErrorCode = ERR_RESULTSET_INVALID
```

ResultSetCount(Command,Response : TSerial)
Donne le nombre d'éléments présents dans un ResultSet.

```
OA = OAManager.getOA(Command.OAID)
If OA.ExistRS(Command.RSID)
        RS = ResultSetManager.getResultSet(Command.RSID)
        Response.COUNT = RS.Count
        Response.Cmd = RESULTSET_COUNT_OK
Else
        Response.Cmd = RESULTSET_COUNT_KO
        Response.ErrorCode = ERR_RESULTSET_INVALID
```

ResultSetFree(Command,Response : TSerial)
Libére de la mémoire le ResultSet spécifié dans Command.

```
OA = OAManager.getOA(Command.OAID)
OA.DeleteRS(Command.RSID)
Response.CMD = RESULTSET_FREE_OK
```

-2- *Exécution asynchrone*

Chaque AsynchWorkerThread va récupérer les messages de la liste AsynchRequest.
Chaque message récupéré est une structure TAsynchRequest contenant toutes les informations nécessaires à l'exécution de la méthode ainsi, que les structures nécessaires pour acceuillir la réponse.
La méthode sera exécutée, et la valeur de retour, stockée dans la structure TAsynchRequest.

TAsynchWorkerThread
Propriétés
AsynchRequest : Tfifo
Liste FIFO pour les requêtes asynchrones.
Méthodes
Constructor Create(AsynchRequest : TFIFO ; AsynchRequestManager : TAsynchRequestManager ; ResultSetManager : TResultSetManager ; OAManager : TOAManager)
Constructeur de la classe.
Seuls les gestionnaires nécessaires sont passés à la classe.

Destroy

Destructeur de la classe.

Execute

Méthode principale du thread.

```
Repeat
        If AsynchRequest.WaitPop(AR, ServerParameters.PollingTime[1])
                OAExec(AR)
Until Terminated
```

OAEXEC(AR : TAsynchRequest)

Exécution de la requête.

```
RS = ResultSetManager.getResultSet(AR.ResultSetID)
AR.State := arsStarted
If OAManager.ExecMethod(AR.Command.OAID, AR.Command.Method, AR.Parameters,
AR.ReturnValue, RS)
        AR.ErrorCode := ERR_NONE
        AR.State := arsTerminated
Else
        AR.ErrorCode := ERR_OAASYNCH_EXECUTE
        AR.ErrorDetail := Exception.Message
        AR.State := arsTerminated
```

[1] L'objet ServerParameters est décrit dans le sous-système de gestion des resources, page 127.

c) Sous-système de gestion des ressources.

Ce sous-système a pour tâche :

- De créer / détruire tous les objets nécessaires au bon fonctionnement du serveur.
- D'analyser le fonctionnement du serveur et de réagir afin d'offrir le meilleur service possible aux utilisateurs.
- De gérer la paramètrisation du serveur.

Le thread principal de l'application paraît idéal pour cette tâche vu que sont fonctionnement est relativement linéaire :

- Création du serveur
- Monitoring du serveur
- Destruction du serveur

La classe TRSOSVRMain va donc encapsuler les fonctionnalités du sous-système de gestion des ressources.

-1- Création des objets

Les objets à créer lors de l'initialisation du serveur sont les suivants :
(cf. Figure III-7 : Schéma général du serveur, Page 60)

Listes:
- Liste FIFO In
- Liste FIFO Out
- Liste AsynchRequest

Threads:
- Worker (n x)
- AsynchWorker (m x)
- Reader (1 x)
- Writer (1 x)

Divers:
- TTCPServer
- TSessionManager
- TResultSetManager
- TOAManager
- TUserManager
- TAsynchManager

-2- Monitoring de l'application

La gestion dynamique du serveur consiste en l'adaptation du nombre de threads servant l'exécution des méthodes synchrones et asynchrones.

Dans un premier temps, on peut imaginer une fonction linéaire dépendant du nombre d'éléments de DataIn (Worker) et de AsynchRequest (AsynchWorker).

Toutefois, ces fonctions doivent êtres bornées par un minimum (ainsi, il y a toujours au moins un thread prêt à réagir) et, par un maximum. (L'utilisation de trop de thread résulte en une chute des performances due à l'augmentation des ressources nécessaires à la gestion de ceux-ci)

Cette adaptation du nombre de thread doit intervenir régulièrement, sans pour autant, surcharger le serveur par des créations et destructions de thread intempestifs.
Une adaptation dynamique toutes les minutes semble raisonnable.

Worker:	N := DataIn.Count / 5 N := Min(N , ServerParameters.MaxWorker) N := Max(N , ServerParameters.MinWorker)
AsynchWorker:	M := AsynchRequest.Count / 5 M := Min(N , ServerParameters.MaxAsynchWorker) M := Max(N , ServerParameters.MinAsynchWorker)

-3- Paramétrisation du serveur

Un certain nombre de paramètres doivent pouvoir être adaptés sans nécessiter une recompilation du serveur. Ces paramètres seront stockés dans un fichier SERVER.INI .
Une classe sera chargée, à l'initialisation de l'application, avec le contenu de ce fichier. Cette classe devra être accessible à l'ensemble du serveur.

TServerParameters
Propriétés
Port : Integer (2000)
Port d'écoute TCP/IP.
PollingTime : DWord (500)
Temps en millisecondes d'attente lors de l'appel à une fonction Win32 bloquante.
MinWorker : Integer (1)
Nombre minimum de WorkerThread.
MaxWorker : Integer (20)
Nombre maximum de WorkerThread.
MinAsynchWorker : Integer (1)
Nombre minimum de AsynchWorkerThread.
MaxAsynchWorker : Integer (20)
Nombre maximum de AsynchWorkerThread.
DynamicMonitoring :DWord (60000)
Temps en millisecondes entre deux adaptations du nombre de threads.
Méthodes
ReadConfig
Lit le fichier SERVER.INI et charge les attributs de la classe

-4- Classe TRSOSVRMain

TRSOSVRMain
Propriétés
TCPServer : TTCPServer
Service réseau
SessionManager : TSessionManager
Gestionnaire de session.
UserManager : TUserManager
Gestionnaire des droits.
OAManager : TOAManager
Gestionnaire des objets d'applications.
AsynchRequestManager : TAsynchRequestManager
Gestionnaire des requêtes asynchrones.
ResultSetManager : TResultSetManager
Gestionnaire de ResultSet.
DataIn : TFIFO
Message entrant.
DataOut : TFIFO
Message sortant.
AsynchRequest : TFIFO
Liste des requêtes asynchrones
ReaderThread : TReaderThread
Thread de gestion des messages entrants (TCP/IP)
WriterThread : TWriterThread
Thread de gestion des messages sortants (TCP/IP)
ListWT : Tlist
Liste des TWorkerThread
ListAWT : Tlist
Liste des TAsynchWorkerThread
Méthodes
Constructor Create
Constructeur de la classe.
Destructor Destroy
Destructeur de la classe.

StartServer

Démarre le serveur proprement dit.

```
TCPServer := TTCPServer.Create

SessionManager := TSessionManager.Create
UserManager := TUserManager.Create
AsynchRequestManager := TAsynchRequestManager.Create
ResultSetManager := TResultSetManager.Create
OAManager := TOAManager.Create(ResultSetManager)

DataIn := TFifo.Create
DataOut := TFifo.Create
AsynchRequest := TFifo.Create

ListWT := TList.Create
ListAWT := TList.Create
For Idx := 1 to ServerParameters.MinWorker Do
        ListWT.Add(TWorker.Create(DataIn, DataOut, AsynchRequest, SessionManager,
        UserManager, OAManager, AsynchRequestManager, ResultSetManager)
For Idx := 1 to ServerParameters.MinAsynchWorker Do
        ListAWT.Add(TAsynchWorker.Create( AsynchRequest, AsynchRequestManager,
        ResultSetManager, OAManager)
ReaderThread := TReaderThread.Create
WriterThread := TwriterThread.Create
```

MonitorServer

Contrôle le fonctionnement du serveur.

```
ServerTerminated := False  //Variable globale destinée à un usage futur pour terminer le
serveur prématurément)
Repeat
        Wait(ServerParameters.DynamicMonitoring)
        Compute N & M (cf Monitoring de l'application, page 126)
        OldWT := ListWT.Count
        While OldWT <> N
                If OldWT < N
                        ListWT.Add(TWorkerThread.Create(...))
                Else
                        ListWT.Index[0].Terminate
                        ListWT.Index[0].Free
        OldAWT := ListAWT.Count
        While OldAWT <> M
                If OldAWT < M
                        ListAWT.Add(TAsynchWorkerThread.Create(...))
                Else
                        ListAWT.Index[0].Terminate
                        ListAWT.Index[0].Free
Until ServerTerminated
```

StopServer

Stoppe le serveur et libère les resources

```
Reader.Terminate
Reader.Free
Writer.Terminate
Writer.Free
While ListAWT.Count > 0 Do
        ListAWT.Index[0].Terminate
        ListAWT.Index[0].Free
While ListWT.Count > 0 Do
        ListWT.Index[0].Terminate
        ListWT.Index[0].Free
TCPServer.Free
DataIn.Free
DataOut.Free
AsynchRequest.Free
```

F. Les OA d'accès aux bases de données

Comme dit, lors de l'analyse fonctionnelle, la méthode d'accès aux bases de données est le Borland Database Engine (BDE).
Cette librairie a l'avantage d'uniformiser les accès à la plupart des SGBD actuels. (Oracle, Sybase, DB2, Interbase, Informix, ODBC, Paradox, MSAccess …)
De plus, le BDE est Thread-Safe.

Chaque instance de l'OA permettra de se connecter à une et une seule base de données, mais permettra, l'exécution de plusieurs requête SQL simultanées.

Les erreurs d'accès aux bases de données sont gérées par chaque méthode. Le serveur RSO aurait pu se charger de remonter simplement l'exception. Le code n'en aurait été que plus simple. Mais, dans ce cas précis, il est intéressant de connaître le code d'erreur natif de la base de données pour permettre une gestion plus fine des erreurs.

TAccessDB
Méthodes
Constructor Create
Constructeur de la classe
Destructor Destroy
destructeur de la classe
ConnectDB (Param : TSerial ; Response : TSerial; ResultSet : TResultSet)
Etablit une connexion avec une base de données.

Param

Driver : String ->	Type de SGBD
ServerName : String ->	Nom du serveur de base de données
UserName : String ->	Login utilisateur
Password : Sgtring ->	Mot de passe de l'utilisateur

ReturnValue

ErrorCode : Integer ->	Code d'erreur de la connexion. 0 si aucune erreur
ErrorDetail : String ->	Message d'erreur si ErrorCode <> 0

Le ResultSet n'est pas employé pour cette méthode.

DisconnectDB (Param : TSerial ; Response : TSerial; ResultSet : TResultSet)
Supprime la connexion active.

Param

/

ReturnValue

ErrorCode: Integer ->	Code d'erreur de la déconnexion. 0 si aucune erreur
ErrorDetail : String ->	Message d'erreur si ErrorCode <> 0

Le ResultSet n'est pas employé pour cette méthode.

PrepareQuery (Param : TSerial ; Response : TSerial; ResultSet : TResultSet)

Prépare une requête SQL et renvoie son identifiant.
Remarque :
la requête SQL peut contenir des paramètres tel que : « Select * From Test Where ID = :ID »

Param

SQL : String ->	Requête SQL

ReturnValue

ID : Integer ->	Identifiant de la requête.
ErrorCode : Integer ->	Code d'erreur du prepare 0 si aucune erreur
ErrorDetail : String ->	Message d'erreur si ErrorCode <> 0

Le ResultSet n'est pas employé pour cette méthode.

ExecuteQuery (Param : TSerial ; Response : TSerial; ResultSet : TResultSet)

Exécute un query précédemment préparé.

Param

ID : Integer ->	Identifiant de la requête
BindParameters : TSerial->	Paramètres de la requête. Chaque nom de variable de la classe TSerial doit correspondre à une variable bind de la requête

ReturnValue

Count : Integer ->	Pour un select, contient le nombre de ligne du ResultSet, pour un autre type de query, le nombre de lignes affectées
FieldName : Tserial ->	Contient le nom des colonnes sous la forme : C1 : String = Nom colonne 1 C2 : String = Nom colonne 2 ...
FieldWidth : Tserial ->	Contient le nom des colonnes sous la forme : C1 : Integer = Taille colonne 1 C2 : Integer = Taille colonne 2 ...
FieldDatatype : Tserial ->	Contient le type des colonnes sous la forme : C1 : Integer = Type colonne 1 C2 : integer = Type colonne 2 ...
ErrorCode : Integer ->	Code d'erreur de l'exécution 0 si aucune erreur
ErrorDetail : String ->	Message d'erreur si ErrorCode <> 0

Dans le cas d'un select, le ResultSet contiendra les ligne de données.

PrepareExecuteQuery (Param : TSerial ; Response : TSerial; ResultSet : TResultSet)

Prépare et exécute une requête SQL.

Param

SQL : String ->	Requête SQL
BindParameters : TSerial->	Paramètres de la requête. Chaque nom de varialble de la classe TSerial doit correspondre à une variable bind de la requête

ReturnValue

ID : Integer ->	Identifiant de la requête SQL
Count : Integer ->	Pour un select, contient le nombre de lignes du ResultSet, pour un autre type de requête SQL, le nombre de lignes affectées.
FieldName : Tserial ->	Contient le nom des colonnes sous la forme : C1 : String = Nom colonne 1 C2 : String = Nom colonne 2 ...
FieldWidth : Tserial ->	Contient le nom des colonnes sous la forme :

	C1 : Integer = Taille colonne 1 C2 : Integer = Taille colonne 2 ...
FieldDatatype : Tserial ->	Contient le type des colonnes sous la forme : C1 : Integer = Type colonne 1 C2 : integer = Type colonne 2 ...
ErrorCode : Integer ->	Code d'erreur de l'exécution 0 si aucune erreur
ErrorDetail : String ->	Message d'erreur si ErrorCode <> 0

Dans le cas d'un select, le ResultSet contiendra les ligne de données.

CloseQuery (Param : TSerial ; Response : TSerial; ResultSet : TResultSet)

Termine un query. Les ResultSets générés lors d'ExecuteQuery ne sont pas détruits, cela reste à la charge du client.

Param

ID : Integer ->	Identifiant query

ReturnValue

ErrorCode : Integer ->	Code d'erreur du close 0 si aucune erreur
ErrorDetail : String ->	Message d'erreur si ErrorCode <> 0

Le ResultSet n'est pas employé pour cette méthode.

Commit (Param : TSerial ; Response : TSerial; ResultSet : TResultSet)

Excécute un commit sur la transaction courante de la base de données.

Param

/

ReturnValue

ErrorCode : Integer ->	Code d'erreur du commit 0 si aucune erreur
ErrorDetail : String ->	Message d'erreur si ErrorCode <> 0

Le ResultSet n'est pas employé pour cette méthode.

Rollback (Param : TSerial ; Response : TSerial; ResultSet : TResultSet)

Excécute un rollback sur la transaction courante de la base de données.

Param

/

ReturnValue

ErrorCode : Integer ->	Code d'erreur du rollback 0 si aucune erreur
ErrorDetail : String ->	Message d'erreur si ErrorCode <> 0

Le ResultSet n'est pas employé pour cette méthode.

V. Implémentation

De par l'ampleur du sujet, et le manque de temps, l'ensemble des fonctionnalités n'a pas été implémenté.

A. Récapitulatif des fonctionnalités implémentées.

1. Service structuration de données (SSD)

Chemin d'accès : .\Source\Service\

Fichier	Classe	Remarques
uDataStruct.Pas	TStringListTS	Implémentée

2. Service réseau (SNT)

Chemin d'accès : .\Source\Service\

Fichier	Classe	Remarques
uTCP.Pas	TSocketElem	Implémentée
	TTCPBase	"
uTCPClient.Pas	TTCPClient	Implémentée
uTCPServer.Pas	TSocketListMessageIn	Implémentée
	TSocketListMessageOut	"
	TTCPEventList	"
	TServerSocketElem	"
	TTCPServer	"

3. Service IPC (SIC)

Chemin d'accès : .\Source\Service\

Fichier	Classe	Remarques
uFIFO.Pas	TFIFO	Implémentée

4. Service sérialisation (SSR)

Chemin d'accès : .\Source\Service\

Fichier	Classe	Remarques
uSerial.Pas	TSerialVariable	Implémentée partiellement. Les types de variables actuellement supportés sont : - String - Integer - TStrings - TSerial
	TSerial	Implémentée

5. Service ResultSet (SST)

Chemin d'accès : .\Source\Service\

Fichier	Classe	Remarques
uResultset.Pas	TResultSet	Implémentée
	TResultsetManager	"

6. Service AsynchRequest (ASY)

Chemin d'accès : .\Source\Service\

Fichier	Classe	Remarques
uAsynchRequest.Pas	TAsynchRequest	Non implémentée
	TAsynchRequestManager	"

7. Service Objet d'application (SOA)

Chemin d'accès : .\Source\Service\

Fichier	Classe	Remarques
uRemoteClass.Pas	TRemote	Implémentée
uSOA.Pas	TLibrary	Implémentée
	TOA	"
	TOAManager	"

8. Service sécurité (SSC)

Chemin d'accès : .\Source\Service\

Fichier	Classe	Remarques
uSecurity.Pas	TUser	Implémentée
	TUserManager	"

9. Service Session (SSM)

Chemin d'accès : .\Source\Service\

Fichier	Classe	Remarques
uSession.Pas	TSession	Implémentée
	TSessionManager	"

10. Module Client

Chemin d'accès : .\Source\RSOCLI\

Fichier	Classe	Remarques
uRSOClient.Pas	TRSOClientBase	Implémentée
	TRSOClient	Implémentée partiellement. La gestion de la sécurité n'est pas présente ; La création d'une session n'est pas asujettie à la vérification d'un utilisateur/ mot de passe.
	TRSORemoteInst	Implémentée
	TRSOQuery	"
	TRSOAsynchQuery	Non implémentée

11. Module Serveur

Chemin d'accès : .\Source\RSOSVR

Fichier	Classe	Remarques
uRSOSVRMain.Pas	TRSOSVRMain	Le monitoring du serveur n'a pas été implémenté. Le nombre de workerThread est fixé à la compilation et n'est pas adapté dynamiquement.
uReaderThread.Pas	TReaderThread	Implémentée
uWriterThread.Pas	TWriterThread	"
uWorkerThread.Pas	TWorkerThread	Implémentée partiellement.La gestion des utilisateurs n'est pas présente. Cette gestion n'étant pas faite, toutes personnes a le droit de créer des sessions et d'utiliser l'ensemble des librairies reconnues par le serveur.
	TAsynchWorkerThread	Non Implémentée
uParameters.Pas	TServerParameters	Cette classe n'a pas été implémentée. Les paramètres gérés par cette classe sont codés au sein de l'application.

12. OA d'accès aux bases de données

Chemin d'accès : .\Source\Library\DBAccess

Fichier	Classe	Remarques
uMain.Pas	TDBAccess	La gestion paramétrique des query SQL n'est pas implémentée. Un query de type : 'SELECT * FROM Table WHERE ID = **:ID**' n'est pas supporté. La classe TSerial ne supportant pas encore l'ensemble des types de données. Les données en provenance de bases de données sont converties sous forme de String.

En résumé, toutes les fonctionnalités du middle-tier ont été implémentées, sauf la gestion des utilisateurs, la gestion des appels asynchrones et le monitoring du serveur.

VI. Exemple

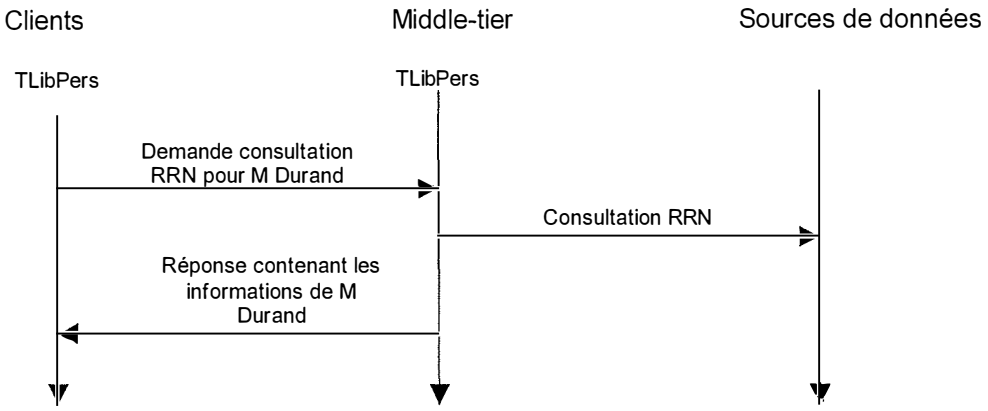
Pour permettre au lecteur de bien comprendre le processus de création d'un objet d'applications et du client qui y accède, un exemple complet a été intégré dans les annexes. (cf. Exemples concrets d'utilisations du middle-tiers page 141)

Celui-ci comprend la rédaction d'un objet d'application et de deux clients. L'un pour des appels synchrones et l'autre pour des appels asynchrones.

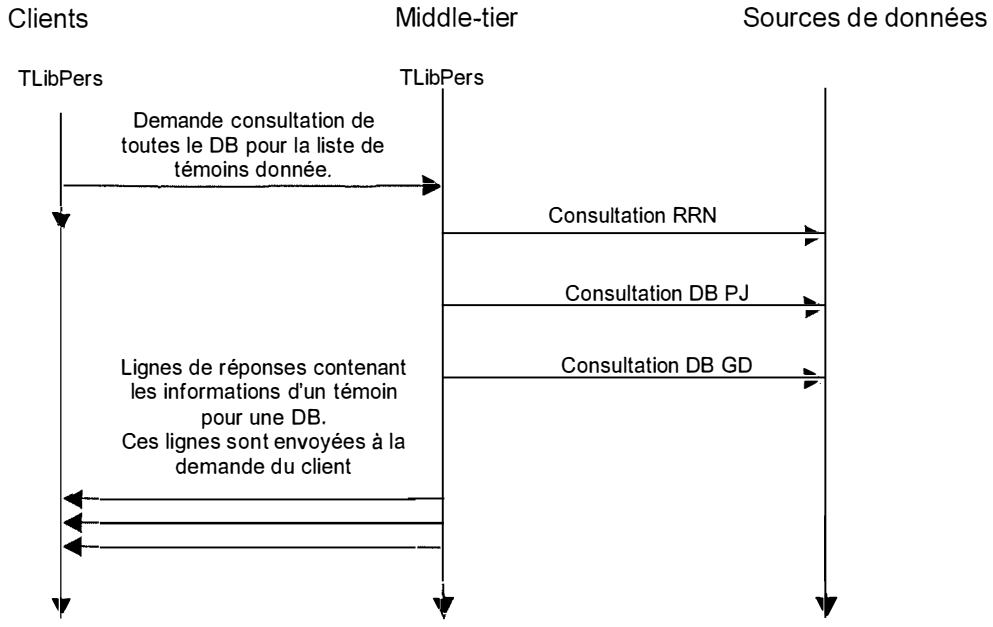
Les cas pris pour cet exemple sont ceux décrits au chapitre des Use Cases. (cf. page 21)

L'objet d'application (TLibPers) permet l'interrogation de diverses bases de données de personnes. (Registre nationale, DB Police Judiciaire, DB Gendarmerie,...)
Les clients implémentent les scénario 1 et 3.

A. Implémentation scénario 1.



B. Implémentation Scénario 3



VII. Performance

Bien que le produit n'ait pas encore été utilisé pour des applications de productions, les premiers tests de performances semblent convaincants.

A. Tests de performance

Pour se rendre compte de la valeur du middle-tier en terme de performance, un test a été élaboré. Celui-ci va permettre de comparer les temps de réponses de RSO par rapport à des middle-ware comme RMI et CORBA.

Le projet consiste en la réalisation d'un serveur implémentant un écho d'une chaîne de caractères et d'un client effectuant des appels à ce serveur. Le client compte le temps mis pour effectuer 100,1000,10000 et 100000 échos. Cela, pour des tailles de chaînes de caractères différentes.

Le lecteur trouvera les détails techniques ainsi que les résultats détaillés de ces tests dans les annexes au chapitre IX.C.1 page 152.

L'analyse des divers graphiques montre que, pour les requêtes comportant des paramètres de petites tailles, CORBA est nettement plus rapide.
Par contre, lorsque la taille des paramètres augmente, le RSO se montre beaucoup plus véloce que CORBA et RMI.
Pour un echo d'une chaine de 30 caractères, le RSO mettra 0,5 ms contre 0,23 ms pour CORBA ; tandis que pour une chaine de 10.000 caratères, le RSO mettra 1,6 ms contre 7.4 ms pour CORBA.
Les rôles s'inversent déjà pour une chaîne de 1.024 caractères, mais, dans de plus faibles proportions.

Les raisons de ces moins bonnes performances pour des paramètres de petites tailles sont les suivantes ; il apparaît que le service Sérialisation (TSerial) est en cause. En effet, ce service est utilisé à outrance et son principe de fonctionnement conduit à énormément de création et destruction d'instances TSerialVariable. Or, ce surplus de travail n'est pas fonction de la taille des paramètres ; ce qui explique les meilleurs temps de réponses obtenus au fur et à mesure que la taille des paramètres augmente.

Une révision de la méthode de fonctionnement de ce service pourrait fortement améliorer les performances et, peut-être, même dépasser les performances du CORBA dans tous les cas.

B. Tests de charges

Ce test a été éléboré en vues de vérifier l'aptitude du serveur RSO à servir l'ensemble des clients de façon équitable.
Il est constitué d'un client simulant un nombre donné de clients . Chaque client simulé exécute sans cesse une requête « Echo »[1] avec une chaîne de caractères de taille donnée.

Le lecteur trouvera les détails techniques ainsi que les résultats détaillés de ces tests dans les annexes au chapitre IX.C.2 page 155.

En moyenne, les temps de réponses du serveur obtenus pour les clients simulés sont semblables.
La différence des divers temps de réponses varie de 0 à quelques millisecondes.
Celle-ci est quasi nulle lorsque la simulation comporte 20 clients. Elle passe à quelques millisecondes pour 100 clients simulés.

[1] La requête Echo est une méthode d'un objet d'application (TLibTest) réalisée en vue de tests du système.

Ces chiffres sont tout à fait acceptables, mais, néanmoins, ils peuvent encore être améliorés. Effectivement, il semble que plus le nombre de client simulé augmente, plus l'écart augmente. Dans une situation idéale, cet écart devrait rester proche de zéro et ne pas évoluer en fonction du nombre de client connecté ! Cela, bien sûr, au détriment du temps de réponse de l'ensemble des clients.

La simulation de plus de 100 clients (en ce qui concerne ce test) n'a pas été faite. Le passage à 200 clients provoque une surcharge du serveur (du moins, sur la machine utilisée pour les tests). La resource CPU étant à 100%, les chiffres obtenus ne sont plus interpretable.

Bien entendu, ces tests ne sont pas la panacée. Beaucoup d'éléments entrent en ligne de compte.

VIII. Conclusion

A. Reste à faire

L'ensemble du produit est fonctionnel. Toutefois, pour pouvoir l'implémenter dans un environnement réel, certains aspects doivent encore être mis en œuvre :

- La gestion des utilisateurs et la gestion des paramètres du serveur doivent être développées.
- Le monitoring du serveur et la gestion des requêtes asynchrones seraient un plus mais ne sont pas strictement nécessaire.
- Quelques bugs subsistent ça et là.

B. Regard sur le produit

Toutes les exigences du produit ont été respectées et les objectifs atteints. De plus, les performances obtenues sont tout à fait correctes, et même, parfois meilleur qu'avec un système comme CORBA.
Il serait toutefois bon de vérifier si, à postériori, certains choix qui ont été faits, étaient les plus judicieux.

Premièrement, le principe de la *signature unique*.
Mon avis est mitigé. Certains pourraient penser que le fait de devoir passer les paramètres avant de faire l'appel à la méthode est quelque chose de lourd. Effectivement, cela peut paraître lourd, mais, cette façon de faire est usage courant en Delphi. Les programmeurs delphi en ont l'habitude. (par exemple, les objets Delphi permettant l'accès aux bases de données fonctionnent de cette façon).
Ce qui est, par contre, plus gênant, est la perte d'informations du point de vue du nombre et du type de paramètres que la méthode accepte.
Une définition d'interface serait dans ce cas la bienvenue. L'aide en ligne de Delphi pourrait alors guider le programmeur.
Dans notre cas, tout ce qui est affiché est le type de notre container (TSerial).
La signature unique impose donc au programmeur de consulter sans cesse la documentation des objets.
Le lecteur trouvera la description d'une éventuelle solution dans le paragraphe réservé aux améliorations possibles.

Deuxièmement, les *ResultSets*.
Cette fonctionnalité n'apporte pas de contraintes supplémentaires.
Les méthodes des objets distants ne sont pas obligées de les utiliser et leur emploi est complètement automatique.
Malgré tout, une faille subsiste. Si un objet distant est mal programmé, il peut générer un resultset gigantesque qui consommerait l'ensemble des ressources mémoires du système.
Pour éviter cela, des quotas doivent être implémentés. Ces quotas seraient administrés par le gestionnaire système qui limiterait la taille des resultset pouvant être générés en fonction de la librairie ou de l'utilisateur.

C. Les améliorations possibles

Les améliorations possibles sont infinies.
Cependant, certaines d'entre elles seraient un véritable plus pour le projet :

1. Générateur d'interface.

Une solution au problème évoqué plus haut de la signature unique serait celle-ci :
A partir d'une définition d'interface, on génèrerait le code permettant le passage d'un appel traditionnel à une méthode vers la méthode utilisée par RSO.

2. Quotas sur les Resultsets

A chaque librairie d'objets distants et/ou à chaque utilisateurs, serait associé un quota. Ce quota indiquerait la taille maximum qu'un resultset peut atteindre. Passé ce quota, l'exécution de la fonction échouerait et le resultset serait détruit.

3. Sauvegarde des ResultSets

Pour permettre néanmoins la création de ResultSets importants, un système de sauvegarde sur disque des Resultsets trop volumineux serait intéressant.

4. Module de gestion

Ce programme permettrait de se connecter à distance vers n'importe quel serveur RSO pour en faire l'administration (Startup, shutdown, ajout d'utilisateur, liste des utilisateurs connectés, ...)

5. Composants DBAware

Le développement de tels composants permettrait d'utiliser le plus simplement du monde la librairie DBAccess. Cela, en utilisant toute la puissance du delphi et de son développement visuel.

IX. Annexes

A. Exemples concrets d'utilisations du middle-tiers

Les exemples présentés au chapitre des Use Cases du mémoire (Page 18) seront pris comme cas pour présenter la manière de coder un objet d'application et ses clients.

1. Implémentation d'un objet d'application

Les deux exemples de client se serviront du même objet d'application. Le but de cet OA est de consulter les diverses bases de données disponibles du point de vue des personnes.
Pour ce faire, une méthode *Consulte* sera implémentée. Cette méthode acceptera comme paramètres une liste de personnes (*ListePers*) ainsi que le nom de la base de données à consulter (*DBName*). Ce nom pourra éventuellement contenir une * si l'on veut consulter l'ensemble des bases de données en une seule fois. Cette façon de faire a été volontairement simplifiée pour des raisons didactiques. En pratique, il serait intéressant de spécifier une liste des bases de données à consulter.

a) Constructeur de l'objet d'application

Ce constructeur est une simple procédure permettant de créer une instance de l'OA. Cette procédure doit, au niveau de la DLL, être exportée.

```
PROCEDURE CREATE_TLIBPERS(Var Inst : TREMOTE; Parameters : TSerial);
Begin
 Inst := TLibPers.Create(Parameters);
End;
```

b) Classe TLibPers

Déclaration

Il est à remarquer que toutes les fonctions ne sont pas accessibles par les clients RSO.
Seule les fonctions déclarées dans la section published pourront être appelées par ceux-ci.

```
TLibTest = Class(TRemote)
  private
    ConsulteRRN(ResultSet : TResultSet);
    ConsulteDBPers(ResultSet : TResultSet);
  public
    Constructor Create(Param : TSerial);
    Destructor Destroy; Override;
  published
    Procedure Consulte(Param : TSerial ; Response : TSerial ; ResultSet : TResultSet);
End;
```

Implémentation

```
constructor TLibPers.Create(Param : TSerial);
begin
 Inherited
// Dans cet exemple, nous n'avons pas besoin de paramètres pour la construction de l'instance
end;

destructor TLibPers.Destroy;
begin
 Inherited
end;


Procedure TLibPers.Consulte(Param: TSerial ; Response : TSerial; ResultSet : TResultSet);
Var DBName : String
begin
  DBName := Param.Variable('DBName').asString;
  if DBName = '*' Then
   Begin
    // A chaque appel à une fonction ConsulteXXX, le ResultSet est augmenté.
    ConsulteRRN(ResultSet)
    ConsulteDBPers(ResultSet)
     .
     .
     .
   End
   else if DBName = 'RRN' Then ConsulteRRN(ResultSet)
   else if …
   .
   .
   .
  // Dans ce cas, nous n'avons pas besoin de la valeur de retour. Toutes l'informations est placées dans le
ResultSet.
  // On pourrait toutefois s'en servir pour renvoyer diverses informations.
  // Ex : Response.Variable('RRNAvailable').asBoolean := false ;
end;


Procedure TLibPers.ConsulteRRN(ResultSet : TResultSet);
Var aRow : TSerial;
begin
  aRow := TSerial.Create ;
  // Exécution du query dans la DB RRN
   While Not Query.eof Do
    Begin
     // Pour chaque ligne renvoyée de la DB RRN, on sauvegarde le résultat dans le ResultSet.
     aRow.Variable('Adresse').String := Query.Field('Adresse').String
     .
     .
     .
     // Sauvegarde d'une ligne de données(TSerial) dans le ResultSet
     ResultSet.SaveData(aRow);
     Query.Next;
    End;
end;
```

```
Procedure TLibPers.ConsulteDBPers(ResultSet : TResultSet);
Var aRow : TSerial;
begin
 aRow := TSerial.Create ;
  // Exécution du query dans la DB Pers
  While Not Query.eof Do
   Begin
    // Pour chaque ligne renvoyée de la DB RRN, on sauvegarde le résultat dans le ResultSet.
    aRow.Variable('Adresse').String := Query.Field('Adresse').String
    .
    .
    .
    // Sauvegarde d'une ligne de données(TSerial) dans le ResultSet
    ResultSet.SaveData(aRow);
    Query.Next;
   End;
end;
```

2. Implémentation d'un client (Appel synchrone)

Voici le cas repris de la page 18 du mémoire.

Ex : L'agent 22, peut ainsi facilement, consulter le registre national de monsieur Durand pour vérifier son identité avant de commencer son audition.

Connexion et création de la session.

```
Var
      ClientRSO          : TRSOClient;
      LibPersonnes       : TRSORemoteInst;
      ConsultPersonnes   : TRSOQuery;
Begin
 // Construction des instances
 ClientRSO := TRSOClient.Create(Nil);
 LibPersonnes := TRSORemoteInst.Create(Nil);
 ConsultPersonnes := TRSOQuery.Create(Nil);

 LibPersonne.RSOClient := ClientRSO;
 ConsultPersonnes.RSORemoteInst := LibPersonnes;

 ClientRSO.CreateSession(Host,Port);
 LibPersonnes.OACreate('LIBPERS.DLL','TLIBPERS');
End ;
```

Cette partie de code n'est pas nécessaire si les composants RSO client sont installé comme composant Delphi.
La création des composants est alors visuelle.

Execution de la demande et traitement du resultat.

```
Var
 ListePersonnes : TStringList;
Begin
 ListePersonnes := Tstringlist.Create;
 ListePersonnes.Add('DURAND/Pierre');
 ConsultePersonnes.Param.Variable('ListePers').asStrings := ListePersonnes ;
 ConsultePersonnes.Param.Variable('DBName').asString := 'RRN' ;
 ConsultePersonnes.Execute('Consulte') ;
 // Le client est bloqué, il attend la réponse du serveur.
 //  Le ResultSet est maintenant chargé avec la liste des personnes correspondant aux critères spécifiés.
 // Tant qu'il y a des lignes disponibles dans le ResultSet
 While Not ConsultePersonne.eof Do
  Begin
    // Récupération des différents champs de la ligne du ResultSet.
    Addresses := ConsultePersonne.ResultSet.Variable('Adresse').asString
    .
    .
    .
    // La ligne suivante du ResultSet devient la ligne courrante.
    ConsultePersonne.ResultSetNext;
  end;
   // Destruction du ResultSet au niveau du serveur. Cette opération n'est pas nécessaire, il sera automatiquement
détruit à la fermeture de l'objet d'application ou de la session.
   ConsultePersonne.ResultSetFree;
end;
```

Fermeture définitive de la session

```
Begin
 LibPersonnes.OAFree;
 RSOCli.Disconnect(dmDisconnect);
end;
```

3. Implémentation d'un client (Appel asynchrone)

Voici le cas repris de la page 19 du mémoire.

Ex : Notre agent 22 a en charge une affaire de meurtre. Après enquête, il se retrouve avec une liste de témoins. Une des premières choses à faire, est de vérifier, au sein des bases de données policières, si l'on connaît quelque chose sur l'un d'eux.
L'agent 22 va donc envoyer au système une requête avec, comme paramètre, la liste des témoins. Après cela, il est 18 heures, notre agent n'a qu'une seul envie, rentrer chez lui. Il éteint son ordinateur comme demandé par la note de service n°32458. Le lendemain matin, il allume son ordinateur, lance son programme de consultation et demande au serveur s'il a la réponse à la requête envoyée la veille. Le serveur ayant eut la nuit entière pour traiter la demande, celui-ci, renvoie les informations que la police dispose sur les témoins.

Connexion et création de la session.

```
Var
      ClientRSO          : TRSOClient;
      LibPersonnes       : TRSORemoteInst;
      ConsultPersonnes   : TRSOAsynchQuery;
Begin
 // Construction des instances
 ClientRSO := TRSOClient.Create(Nil);
 LibPersonnes := TRSORemoteInst.Create(Nil);
 ConsultPersonnes := TRSOQuery.Create(Nil);

 LibPersonne.RSOClient := ClientRSO;
 ConsultPersonnes.RSORemoteInst := LibPersonnes;

 ClientRSO.CreateSession(Host,Port);
 LibPersonnes.OACreate('LIBPERS.DLL','TLIBPERS');
End ;
```

Cette partie de code n'est pas nécessaire si les composants RSO client sont installé comme composant Delphi.
La création des composants est alors visuelle.

Envoi de la liste des personnes à consulter au serveur

```
Var
 ListePersonnes : TStringList;
Begin
 ListePersonnes := Tstringlist.Create;
 ListePersonnes.Add('COYETTE/Laurent');
 ListePersonnes.Add('Arrotin/Pascal') ;
 ConsultePersonnes.Param.Variable('ListePers').asStrings := ListePersonnes ;
 ConsultePersonnes.Param.Variable('DBName').asString := '*' ;
 ConsultePersonnes.Execute('Consulte') ;
End ;
```

Exécution de la méthode 'Consulte' de TLibPers

A partir de ce momment, la demande est faite, le client peut se déconnecter.

Détachement du client

```
Begin
 // Sauvegarde du n° de requête (ConsultePersonnes.ReqID) pour récupérer ultérieurement le résultat.
 // Sauvegarde du n° de session (RSOClient.Session) pour se reconnecter ultérieurement à la même session.
 RSOCli.Disconnect(dmDetach);
End;
```

Reconnexion du serveur à une session existante

```
Begin
 // Récupération du numéro de session précédemment sauvegardé
 RSOCli.Session := Session
 RSOCli.AttachSession(Host,Port)
End ;
```

Test de la disponibilité de la réponse

```
Begin
 // Récupération du numéro de requête sauvegardé
 ConsultePersonnes.ReqID := Requête
 If ConsultePersonne.CheckState = aqsTerminated
 Then // La requête est terminée, on peut la traiter.
End ;
```

Traitement du résultat de la requête.

```
Var
 DBName : String;
Begin
  // Le ResultSet est positionné sur sa première ligne.
 ConsultePersonne.ResultSetFirst;
 // Tant qu'il y a des lignes disponibles dans le ResultSet
 While Not ConsultePersonne.eof Do
  Begin
    DBName := ConsultePersonne.ResultSet.Variable('DBName').asString;
    if  DBName = 'RRN' Then
     Begin
       // Récupération des différents champs de la ligne du ResultSet propre à la DB RRN.
     End
    Else if DBName = 'PERS'
     Begin
       // Récupération des différents champs de la ligne du ResultSet propre à la DB PERS.
     End
    .
    .
    .
    // La ligne suivante du ResultSet devient la ligne courrante.
    ConsultePersonne.ResultSetNext;
  end;
    // Destruction du ResultSet au niveau du serveur. Cette opération n'est pas nécessaire, il sera
automatiquement détruit à la fermeture de l'objet d'application ou de la session.
    ConsultePersonne.ResultSetFree;
end;
```

Fermeture définitive de la session

```
Begin
 LibPersonnes.OAFree;
 RSOCli.Disconnect(dmDisconnect);
end;
```

B. Sources

L 'ensemble du code source du projet RSO représente 10.000 lignes de code. Dans un souci d'économie de papier, les sources n'ont donc pas été imprimées. Elles sont toutefois disponibles sur la disquette fournie avec le mémoire.

1. Arborescence du projet

```
+---Exe                          Exécutables.
|   \---Library                  Librairies d'objets d'applications.
+---Obj                          Ensemble des OBJ générés lors de la compilation.
|   +---CLIENTS
|   |   +---CLIENTTEST
|   |   \---LOADCLIENT
|   +---LIBRARY
|   |   +---DBACCESS
|   |   \---LIBTEST
|   +---RSOMGR
|   \---RSOSVR
\---Sources                      Sources du projet.
    +---CLIENTS                  Sources des divers clients de tests.
    |   +---ClientTest
    |   +---LoadClient
    |   \---RSOCLI               Sources du module client permettant l'accès au RSO Serveur.
    +---COMMON                   Diverses sources communes à l'ensemble des applications.
    +---LIBRARY                  Sources des librairies d'OA.
    |   +---DBACCESS
    |   \---LIBTEST
    +---RSOSVR                   Sources du module serveur.
    +---SERVICES                 Sources des divers services.
    \---TOOLS                    Sources des outils third party utilisés.
        +---JCL
        |   +---Packages
        |   \---Source
```

2. Description des fichiers

a) Fichiers communs

Chemin : RSO\sources\common

Ces fichiers sont accessibles à l'ensemble des applications.

Fichiers	Descriptions
uCommon.pas	Fonctions d'interêt général, constantes, ...
uDebug.pas	Fonctions de déboguage
uSysTools.pas	Fonctions système.

b) Outils

Chemin : RSO\sources\tools

Fichiers	Descriptions
JCL*.*	Librairie de fonctions *Jedi Code Library* (www.jedi.org) Collection de fonctions distribuée sous licence MPL (Mozilla Public License).

c) Services

Chemin : RSO\sources\Services

Implémentations des divers services présentés dans le mémoire.

Fichiers	Descriptions
uDataStruct.Pas	Gestion de structures de données.
uTCP.Pas	TCP/IP de base
uTCPClient.Pas	TCP/IP pour les clients.
uTCPServer.Pas	TCP/IP pour le serveur
uFIFO.Pas	Gestion de piles
uSerial.Pas	Gestion des containers (TSerial)
uResultset.Pas	Gestion des ResultSets
uRemoteClass.Pas	Classe de base pour les OA
uSOA.Pas	Gestion des OA
uSecurity.Pas	Gestion de la sécurité
uSession.Pas	Gestion des sessions

d) Module Client

Chemin : RSO\sources\Clients\RSOClient)

Implémentation des composants d'accès au RSO serveur.

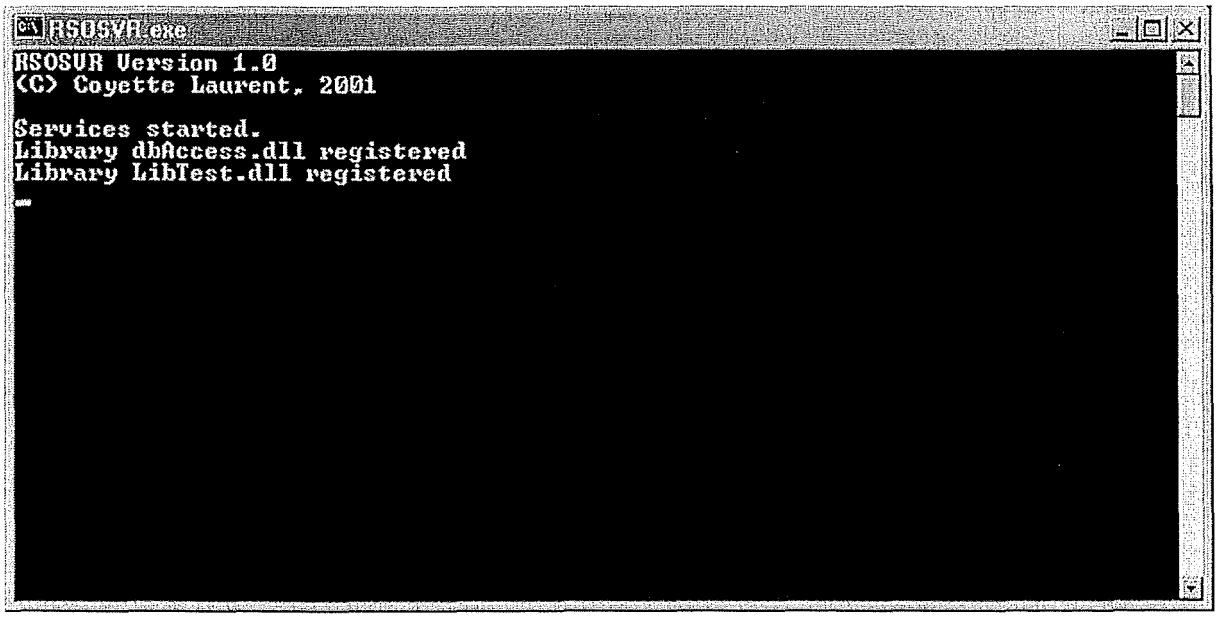
Fichiers	Descriptions
uRSOClient.Pas	Composant RSO clients

e) Module Serveur

Chemin : RSO\Sources\RSOSVR

Implémentation du RSO Serveur

Fichiers	Descriptions
RsoSvr.dpr	Définition du projet
uRSOSVRMain.Pas	Gestion des resources
uReaderThread.Pas	Gestion des communications entrantes
uWriterThread.Pas	Gestion des communications sortantes
uWorkerThread.Pas	Gestion des requêtes



f) Librairie DBAccess

Chemin : RSO\Sources\Libraryi\DBAccess

Implémentation de la librairie (OA) d'accès aux bases de données en utilisant le BDE.

Fichiers	Descriptions
DBAccess.dpr	Définition du projet
uDBAMain.Pas	Implémentation de la classe TDBAccess

g) Librairie LibTest

Chemin : RSO\Sources\Library\LibTest

Implémentation d'une librairie(OA) de test. La seule fonction de cette librairie est Echo renvoyant une chaîne de caractères passée en paramètre.

Fichiers	Descriptions
LibTest.dpr	Définition du projet
uLibTestMain.Pas	Implémentation de la classe TLibTest

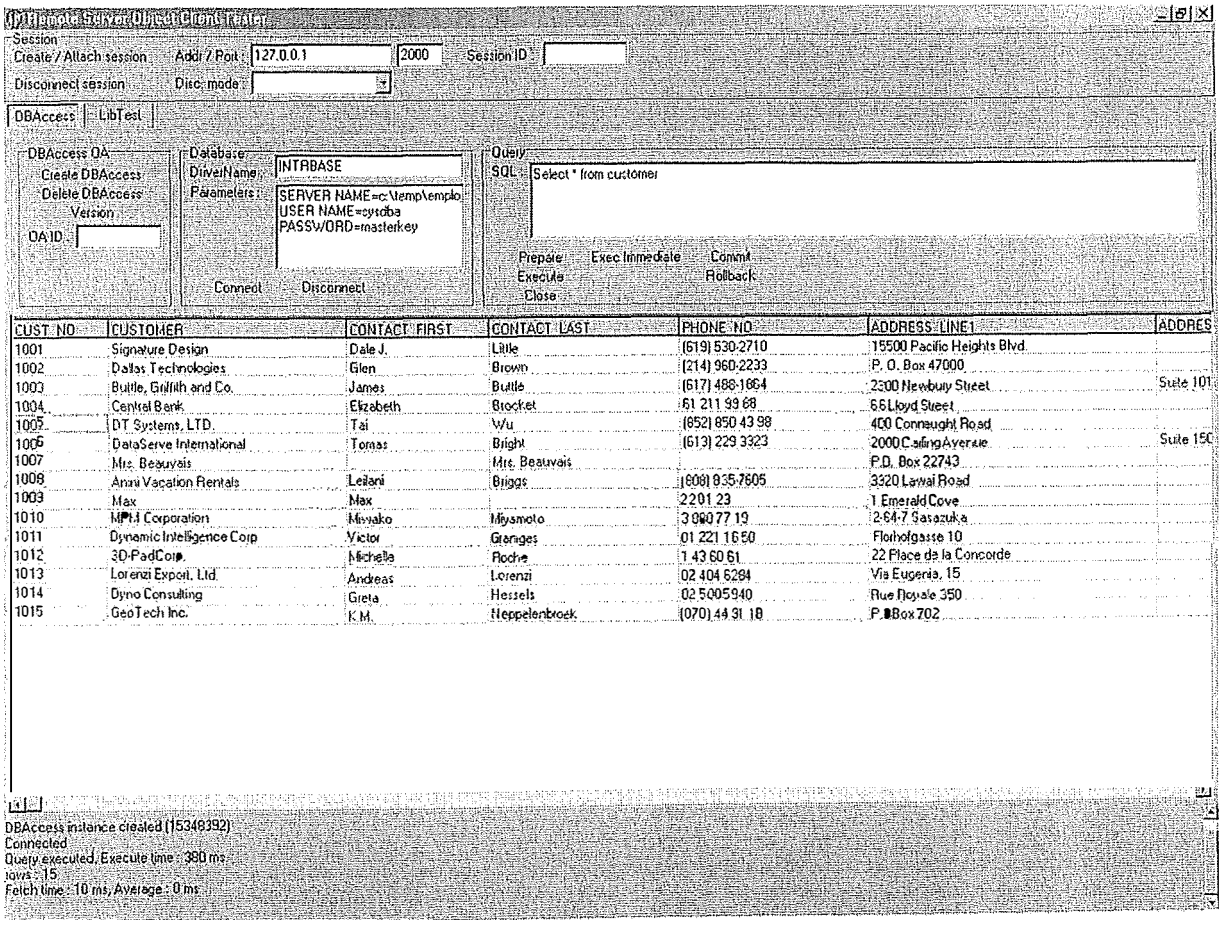
h) ClientTest

Chemin : RSO\Sources\Clients\ClientTest

Programme élaboré dans le cadre des tests du RSO serveur.
Cette application permet de :

- créer et détruire une session
- s'attacher et se détacher d'une session
- créer et détruire des OA TLibTest et TDBAccess
- Utiliser les OA TLibTest et TDBAccess.

Fichiers	Descriptions
ClientTest.dpr	Définition du projet
fMain.Pas	Fenêtre principal de ClientTest

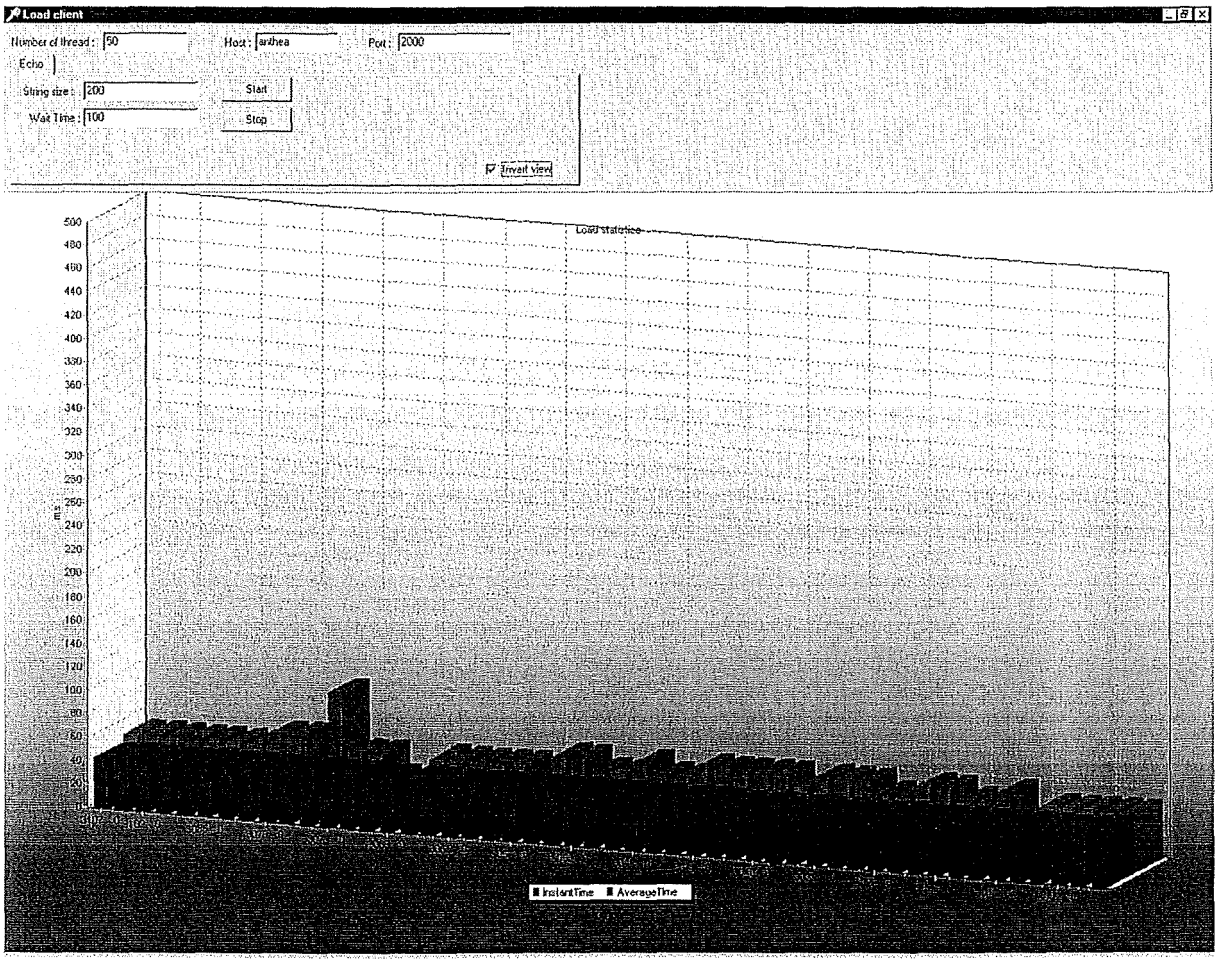


i) LoadClient

Chemin : RSO\Sources\Clients\LoadClient

Programme élaboré dans le cadre des tests du RSO serveur.
Cette application simule un nombre donné de clients en vue de charger le serveur. Chacun de ces clients effectue une requête Echo avec une chaîne de caractères de taille donnée et ce, à intervalle régulier.
Les temps de réponses instantanés et moyens sont affichés pour chaque client.

Fichiers	Descriptions
LoadClient.dpr	Définition du projet
uMain.pas	Fenêtre principal de LoadClient
uSendPacket	Gestion du thread effectuant les requêtes



C. Les tests

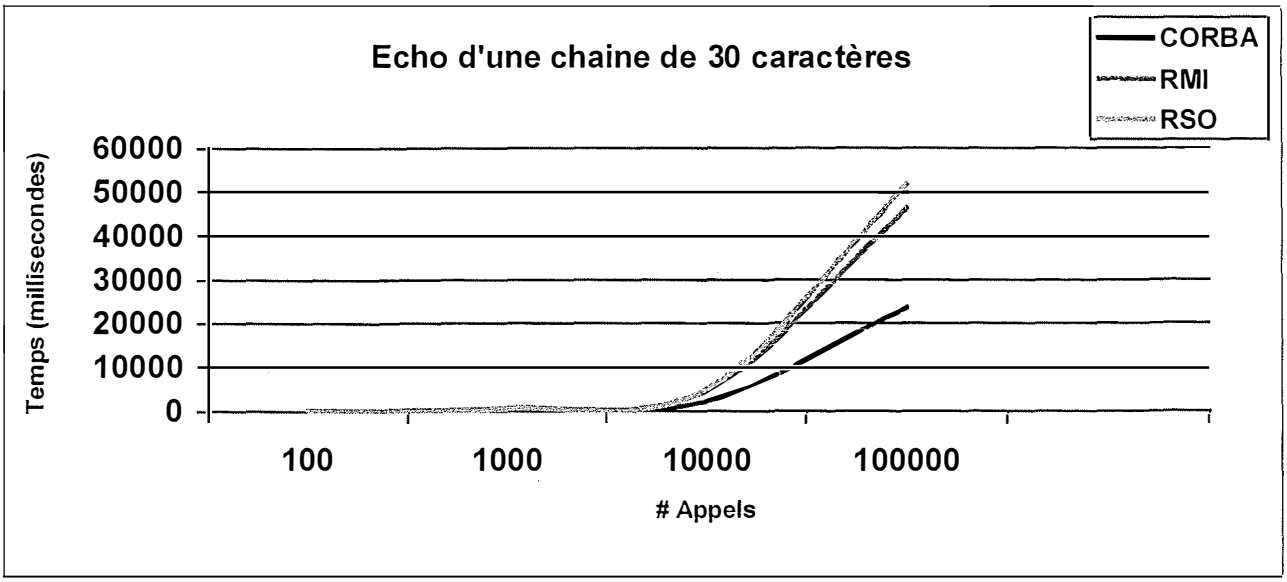
1. Tests de performance.

Pour situer le projet par rapport aux produits existants, un petit test a été élaboré.
Ce test permet de comparer les temps de réponses du RSO serveur par rapport à CORBA et RMI.
Il consiste en l'appel d'une même fonction un certain nombre de fois et de totaliser le temps mis entre l'envoi de la requête au serveur et la réception de la réponse.
La méthode appelée est *Echo* de TLibTest.
Trois systèmes remplissant ces fonctionnalités ont été réalisés en utilisant Corba, RMI et RSO.
La version Corba a été développée en Delphi / VisiBroker (Corba).
La version RMI a été développée en Java (Sun, JDK 1.3.1)

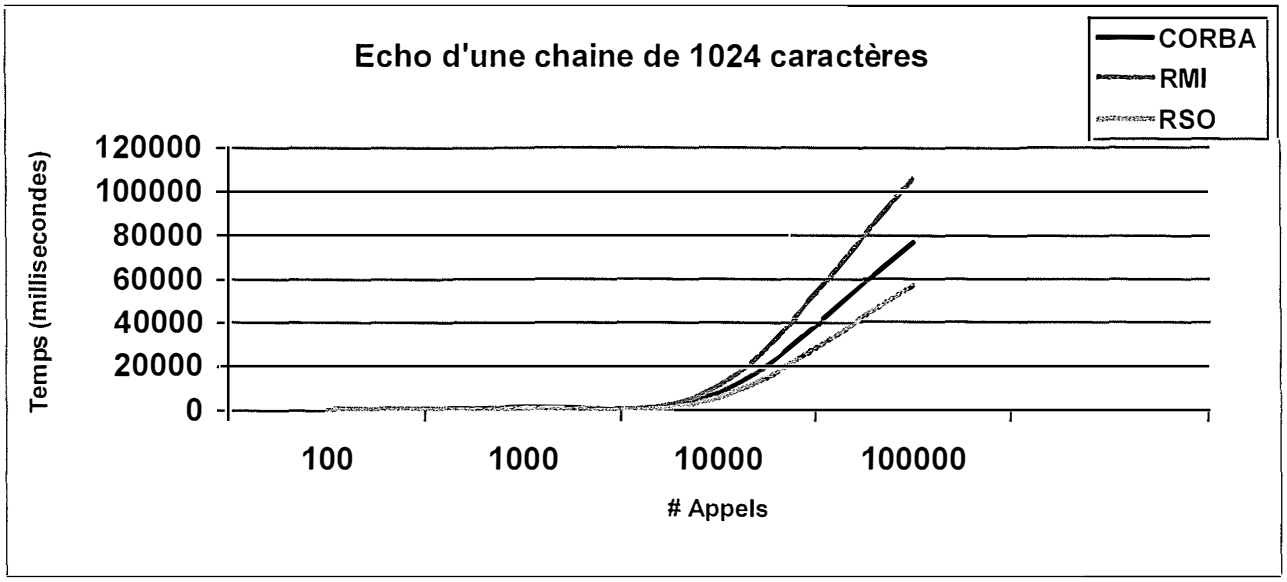
Le test de chaque système comprenait 100,1000,10.000 et 100.000 appels consécutifs. Ces appels ont étés réalisés, successivement, avec une longueur de chaîne (pour l'echo) de 30 bytes, 1024 bytes et 10.000 bytes.

Les sources de ces trois systèmes sont disponibles sur la disquette fournie avec le mémoire dans le répertoire RSOTiming

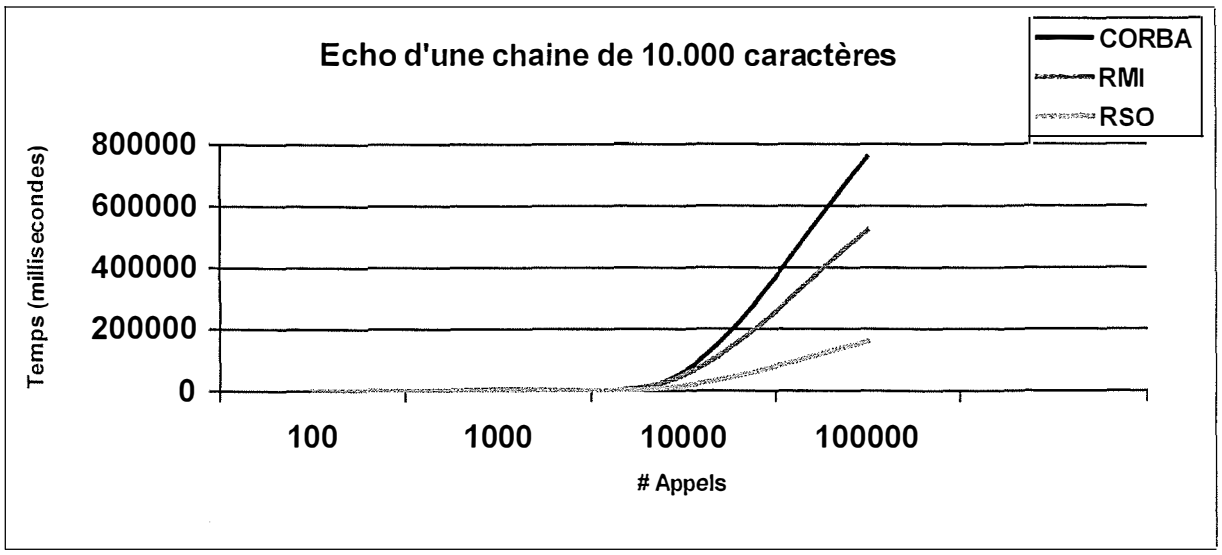
Résultat des tests



# Call	CORBA	RMI	RSO
100	30	100	61
1000	241	731	520
10.000	2.363	4.837	5.178
100.000	23.704	46.377	51.964



# Call	CORBA	RMI	RSO
100	80	181	50
1000	761	1.352	561
10.000	7.671	10.805	5.578
100.000	76.981	105.963	57.002

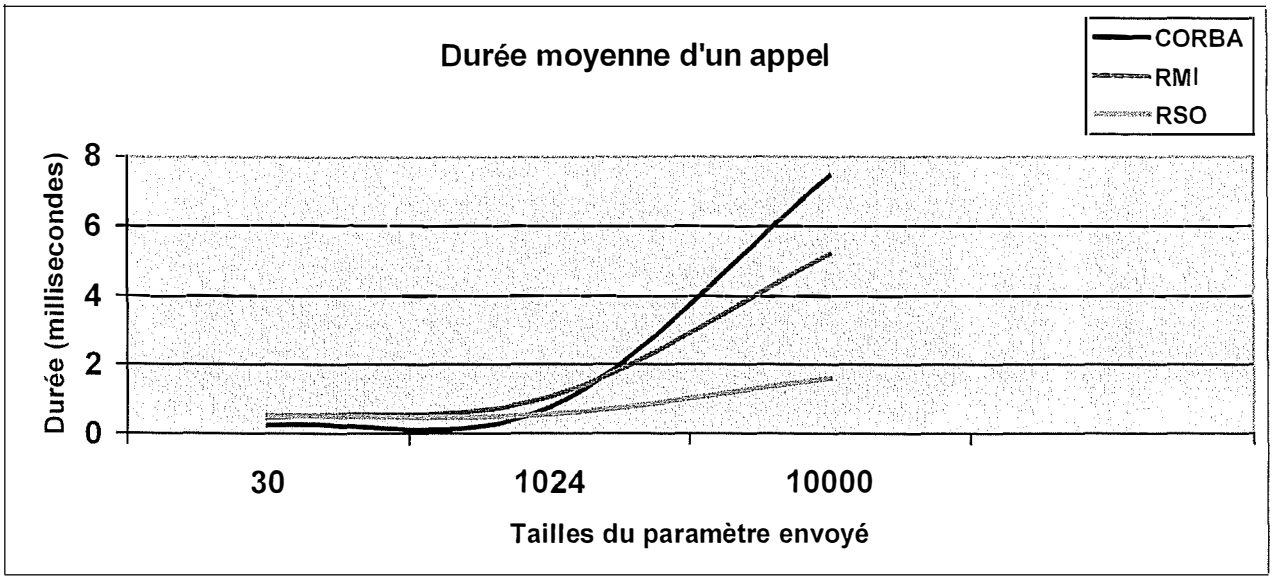


# Call	CORBA	RMI	RSO
100	631	642	140
1.000	6.269	5.518	1.513
10.000	63.371	52.145	16.103
100.000	758.671	519.076	160.671

En analysant ces graphiques, on se rend compte que, pour des appels avec des paramètres de petites tailles, le CORBA est nettement plus rapide que le RSO. Au fur et à mesure que la taille des paramètres augmente, la tendance s'inverse fortement.
Après une analyse détaillée, il s'avère que la classe TSerial, utilisée à outrance dans l'ensemble du projet RSO présente des lacunes en terme de performance. Cela entraîne un overhead important qui ne dépend pas de la taille des paramètres. Cela explique les moins bonnes performances obtenues lors du test avec 30 caractères.

En révisant complètement cette classe, des gains de performance de l'ordre de 200% ne seraient pas impossibles.

Ces tests montrent que le RSO s'avère plus constant que les deux autres systèmes.
Le graphique suivant montre l'évolution de la durée d'un appel à la fonction *echo* en fonction de la taille du paramètre envoyé. La courbe du RSO évolue beaucoup moins vite que celle du CORBA et du RMI.



# Caractères	CORBA	RMI	RSO
30	0.2370	.4684	.5195
1.024	.7695	1.0648	.5687
10.000	7.4612	5.1969	1.6060

2. Tests de charges

Un autre programme de tests a été élaboré en vue de vérifier que les clients sont servis de façon équitable.
Ce programme simule un nombre donné de client. Chacun de ces clients simulés, envoi une requête Echo avec une taille de paramètre donné, attende la réponse et recommence le processus jusqu'à ce que le test soit arrêté. Le nombre de requête émise, le temps mis ainsi que le temps moyen pour effectuer une requête sont notés dans un fichier.
Ce test a été réalisé successivement pour des paramètres de 30, 1024 et 10000 caractères. Et ce, pour 20 et 100 clients simultanés.
Ces tests ont été répétés plusieurs fois pour vérifier la constance des résultats.

Résultat des tests

Les temps pour chaque client étant très similaires, seul un résumé a été intégré dans ce document.

Nombre de clients	Taille du paramètre	Temps de réponse minimum (ms)	Temps de réponse maximum (ms)	Ecart (ms)
20	30	34,38	34,45	0,07
20	1.024	39,22	39,25	0,03
20	10.000	64,84	64,96	0,12
100	30	283,21	286,43	3,22
100	1.024	292,31	295,43	3,12
100	10.000	368,22	374,72	6,5

Pour un nombre de clients donné, les clients sont plus ou moins servis de façon équitable.

X. Bibliographie.

- Kenneth P. Birman, Building Secure and Reliable Network Applications, Department of Computer Science. Cornell University, 1995
- Jenkins Andy, When is CORBA Worth Investigating, CSE 588: Network Systems (Jun 1999)
- The Common Object Request Broker: Architecture and Specification, Object Management Group, 1991 – 2001
- Writing Corba Applications, Delphi 5 documentation.
- Java Builder 5 documentation.
- Java Sun Documentation (JDK 1.3.1)

Beaucoup d'informations ont étés trouvées sur Internet par l'intermédiaire de forum, newsgroup, …